VISIT…

SECOND EDITION

ESSENTIAL MATHEMATICS FOR GAMES & INTERACTIVE APPLICATIONS

A PROGRAMMER'S GUIDE

JAMES M. VAN VERTH
LARS M. BISHOP

MORGAN KAUFMANN

This excellent volume is unique in that it covers not only the basic techniques of computer graphics and game development, but also provides a thorough and rigorous—yet very readable—treatment of the underlying mathematics. Fledgling graphics and games developers will find it a valuable introduction; experienced developers will find it an invaluable reference. Everything is here, from the detailed numeric issues of IEEE floating point notation, to the correct way to use quaternions and spherical linear interpolation to represent orientation, to the mathematics of collision detection and rigid-body dynamics.

—David Luebke, University of Virginia,
co-author of *Level of Detail for 3D Graphics*

When it comes to software development for games or virtual reality, you cannot escape the mathematics. The best performance comes not from superfast processors and terabytes of memory, but from well-chosen algorithms. With this in mind, the techniques most useful for developing production-quality computer graphics for Hollywood blockbusters are not the best choice for interactive applications. When rendering times are measured in milliseconds rather than hours, you need an entirely different perspective.

Essential Mathematics for Games and Interactive Applications provides this perspective. While the mathematics are rigorous and perhaps challenging at times, Van Verth and Bishop provide the context for understanding the algorithms and data structures needed to bring games and VR applications to life. This may not be the only book you will ever need for games and VR software development, but it will certainly provide an excellent framework for developing robust and fast applications.

—Ian Ashdown, President, ByHeart Consultants Limited

With *Essential Mathematics for Games and Interactive Applications,* Van Verth and Bishop have provided invaluable assistance for professional game developers looking to shore up weaknesses in their mathematical training. Even if you never intend to write a renderer or tune a physics engine, this book provides the mathematical and conceptual grounding needed to understand many of the key concepts in rendering, simulation, and animation.

—Dave Weinstein, Microsoft, Red Storm Entertainment

Geometry, trigonometry, linear algebra, and calculus are all essential tools for 3D graphics. Mathematics courses in these subjects cover too much ground, while at the same time glossing over the bread-and-butter essentials for 3D graphics programmers. In *Essential Mathematics for Games and Interactive Applications,* Van Verth and Bishop bring just the right level of mathematics out of the trenches of professional game development. This book provides an accessible and solid mathematical foundation for interactive graphics programmers. If you are working in the area of 3D games, this book is a "must have."

—Jonathan Cohen, Department of Computer Science,
Johns Hopkins University,
co-author of *Level of Detail for 3D Graphics*

It's the book with all the math you need for games.

—Neil Kirby, Bell Labs

As games become ever more sophisticated, mathematics and technical programming skills become increasingly important to have in your toolbox. *Essential Math* provides a solid foundation in many critical areas. You will find many topics covered in detail: from linear algebra to calculus, from physics to rasterization. Some of this will be review material, but you will undoubtedly learn something new and, most importantly, something useful.

—Erin Catto, Blizzard Entertainment

This page intentionally left blank

Essential Mathematics for Games and Interactive Applications

A Programmer's Guide

Second Edition

This page intentionally left blank

Essential Mathematics for Games and Interactive Applications

A Programmer's Guide

Second Edition

James M. Van Verth

Lars M. Bishop

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

Acquisitions Editor	*Laura Lewin*
Assistant Editor	*Chris Simpson*
Publishing Services Manager	*George Morrison*
Senior Production Manager	*Paul Gottehrer*
Cover Designer	*Joanne Blank*
Composition	*diacriTech*
Interior printer	*RR Donnelley*
Cover printer	*Phoenix Color*

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

This book is printed on acid-free paper. ♾

Copyright © 2008 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, E-mail: permissions@elsevier.com. You may also complete your request online via the Elsevier homepage (http://elsevier.com), by selecting "Support & Contact" then "Copyright and Permission" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data

APPLICATIONS SUBMITTED
ISBN: 978-0-12-374297-1
ISBN: 978-0-12-374298-8 (CD-ROM)

For information on all Morgan Kaufmann publications,
visit our Web site at *www.mkp.com* or *www.books.elsevier.com*

Printed in The United States of America
08 09 10 11 12 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID International Sabre Foundation

Dedications

To Mur and Fiona, for allowing me to slay the monster one more time. —Jim

To Jen, who constantly helps me get everything done; and to Nadia and Tasha, who constantly help me avoid getting any of it done on time. —Lars

About the Authors

James M. Van Verth is an OpenGL Software Engineer at NVIDIA, where he works on device drivers for NVIDIA GPUs. Prior to that, he was a founding member of Red Storm Entertainment, where he was a lead engineer for eight years. For the past nine years he also has been a regular speaker at the Game Developers Conference, teaching the all-day tutorials "Math for Game Programmers" and "Physics for Game Programmers," on which this book is based. His background includes a B.A. in Math/Computer Science from Dartmouth College, an M.S. in Computer Science from the State University of New York at Buffalo, and an M.S. in Computer Science from the University of North Carolina at Chapel Hill.

Lars M. Bishop is an engineer in the Handheld Developer Technologies group at NVIDIA. Prior to joining NVIDIA, Lars was the Chief Technology Officer at Numerical Design Limited, leading the development of the Gamebryo3D cross-platform game engine. He received a BS in Math/Computer Science from Brown University and an MS in Computer Science from the University of North Carolina at Chapel Hill. His outside interests include photography, drumming, and playing bass guitar.

Contents

Preface xix

Introduction xxiii

Chapter 1
Real-World Computer Number Representation 1

1.1 Introduction 1
1.2 Representing Real Numbers 2
1.2.1 Approximations 2
1.2.2 Precision and Error 3
1.3 Floating-Point Numbers 4
1.3.1 Review: Scientific Notation 4
1.3.2 A Restricted Scientific Notation 5
1.4 Binary "Scientific Notation" 6
1.5 IEEE 754 Floating-Point Standard 9
1.5.1 Basic Representation 9
1.5.2 Range and Precision 11
1.5.3 Arithmetic Operations 13
1.5.4 Special Values 16
1.5.5 Very Small Values 19
1.5.6 Catastrophic Cancelation 22
1.5.7 Double Precision 24
1.6 Real-World Floating-Point 25
1.6.1 Internal FPU Precision 25
1.6.2 Performance 26
1.6.3 IEEE Specification Compliance 29
1.6.4 Graphics Processing Units and Half-Precision Floating-Point Formats 31
1.7 Code 32
1.8 Chapter Summary 33

CHAPTER 2
VECTORS AND POINTS 35

2.1 **Introduction 35**
2.2 **Vectors 36**
2.2.1 Geometric Vectors 36
2.2.2 Linear Combinations 39
2.2.3 Vector Representation 40
2.2.4 Basic Vector Class Implementation 42
2.2.5 Vector Length 44
2.2.6 Dot Product 47
2.2.7 Gram-Schmidt Orthogonalization 51
2.2.8 Cross Product 53
2.2.9 Triple Products 56
2.2.10 Real Vector Spaces 59
2.2.11 Basis Vectors 62
2.3 **Points 63**
2.3.1 Points as Geometry 64
2.3.2 Affine Spaces 66
2.3.3 Affine Combinations 68
2.3.4 Point Implementation 70
2.3.5 Polar and Spherical Coordinates 72
2.4 **Lines 75**
2.4.1 Definition 75
2.4.2 Parameterized Lines 76
2.4.3 Generalized Line Equation 77
2.4.4 Collinear Points 79
2.5 **Planes 80**
2.5.1 Parameterized Planes 80
2.5.2 Generalized Plane Equation 80
2.5.3 Coplanar Points 82
2.6 **Polygons and Triangles 82**
2.7 **Chapter Summary 86**

CHAPTER 3
MATRICES AND LINEAR TRANSFORMATIONS 87

3.1 **Introduction 87**
3.2 **Matrices 88**
3.2.1 Introduction to Matrices 88
3.2.2 Simple Operations 90
3.2.3 Vector Representation 92
3.2.4 Block Matrices 92
3.2.5 Matrix Product 94
3.2.6 Identity Matrix 96

3.2.7 Performing Vector Operations with Matrices 97
3.2.8 Implementation 98
3.3 **Linear Transformations 101**
3.3.1 Definitions 101
3.3.2 Null Space and Range 103
3.3.3 Linear Transformations and Basis Vectors 104
3.3.4 Matrices and Linear Transformations 106
3.3.5 Combining Linear Transformations 108
3.4 **Systems of Linear Equations 110**
3.4.1 Definition 110
3.4.2 Solving Linear Systems 112
3.4.3 Gaussian Elimination 113
3.5 **Matrix Inverse 117**
3.5.1 Definition 117
3.5.2 Simple Inverses 120
3.6 **Determinant 121**
3.6.1 Definition 121
3.6.2 Computing the Determinant 123
3.6.3 Determinants and Elementary Row Operations 126
3.6.4 Adjoint Matrix and Inverse 128
3.7 **Eigenvalues and Eigenvectors 129**
3.8 **Chapter Summary 130**

CHAPTER 4
AFFINE TRANSFORMATIONS 133

4.1 **Introduction 133**
4.2 **Affine Transformations 134**
4.2.1 Matrix Definition 134
4.2.2 Formal Definition 136
4.2.3 Formal Representation 138
4.3 **Standard Affine Transformations 139**
4.3.1 Translation 139
4.3.2 Rotation 141
4.3.3 Scaling 150
4.3.4 Reflection 151
4.3.5 Shear 154
4.3.6 Applying an Affine Transformation Around an Arbitrary Point 156
4.3.7 Transforming Plane Normals 158
4.4 **Using Affine Transformations 159**
4.4.1 Manipulation of Game Objects 159
4.4.2 Matrix Decomposition 164
4.4.3 Avoiding Matrix Decomposition 166

4.5 **Object Hierarchies 169**
4.6 **Chapter Summary 171**

CHAPTER 5
ORIENTATION REPRESENTATION 173

5.1 **Introduction 173**
5.2 **Rotation Matrices 174**
5.3 **Fixed and Euler Angles 174**
5.3.1 Definition 174
5.3.2 Format Conversion 177
5.3.3 Concatenation 178
5.3.4 Vector Rotation 178
5.3.5 Other Issues 179
5.4 **Axis–Angle Representation 181**
5.4.1 Definition 181
5.4.2 Format Conversion 182
5.4.3 Concatenation 184
5.4.4 Vector Rotation 184
5.4.5 Axis–Angle Summary 185
5.5 **Quaternions 185**
5.5.1 Definition 185
5.5.2 Quaternions as Rotations 186
5.5.3 Addition and Scalar Multiplication 187
5.5.4 Negation 188
5.5.5 Magnitude and Normalization 188
5.5.6 Dot Product 189
5.5.7 Format Conversion 189
5.5.8 Concatenation 193
5.5.9 Identity and Inverse 195
5.5.10 Vector Rotation 197
5.5.11 Shortest Path of Rotation 199
5.5.12 Quaternions and Transformations 200
5.6 **Chapter Summary 201**

CHAPTER 6
VIEWING AND PROJECTION 203

6.1 **Introduction 203**
6.2 **View Frame and View Transformation 205**
6.2.1 Defining a Virtual Camera 205
6.2.2 Constructing the View-to-World Transformation 206
6.2.3 Controlling the Camera 208
6.2.4 Constructing the World-to-View Transformation 211

6.3 **Projective Transformation 212**
6.3.1 Definition 212
6.3.2 Normalized Device Coordinates 216
6.3.3 View Frustum 216
6.3.4 Homogeneous Coordinates 220
6.3.5 Perspective Projection 221
6.3.6 Oblique Perspective 228
6.3.7 Orthographic Parallel Projection 231
6.3.8 Oblique Parallel Projection 232
6.4 **Culling and Clipping 235**
6.4.1 Why Cull or Clip? 235
6.4.2 Culling 238
6.4.3 General Plane Clipping 239
6.4.4 Homogeneous Clipping 244
6.5 **Screen Transformation 246**
6.5.1 Pixel Aspect Ratio 248
6.6 **Picking 249**
6.7 **Management of Viewing Transformations 252**
6.8 **Chapter Summary 254**

CHAPTER 7
GEOMETRY AND PROGRAMMABLE SHADING 255

7.1 **Introduction 255**
7.2 **Color Representation 257**
7.2.1 RGB Color Model 257
7.2.2 Colors as "Vectors" 257
7.2.3 Color Range Limitation 258
7.2.4 Operations on Colors 259
7.2.5 Alpha Values 260
7.2.6 Color Storage Formats 264
7.3 **Points and Vertices 266**
7.3.1 Per-Vertex Attributes 266
7.3.2 An Object's Vertices 267
7.4 **Surface Representation 270**
7.4.1 Vertices and Surface Ambiguity 270
7.4.2 Triangles 271
7.4.3 Connecting Vertices into Triangles 271
7.4.4 Drawing Geometry 274
7.5 **Rendering Pipeline 275**
7.5.1 Fixed-Function versus Programmable Pipelines 277
7.6 **Shaders 278**
7.6.1 Using Shaders to Move from Vertex to Triangle to Fragment 278

7.6.2 Shader Input and Output Values 279
7.6.3 Shader Operations and Language Constructs 280
7.7 Vertex Shaders 280
7.7.1 Vertex Shader Inputs 280
7.7.2 Vertex Shader Outputs 281
7.7.3 Basic Vertex Shaders 282
7.7.4 Linking Vertex and Fragment Shaders 282
7.8 Fragment Shaders 283
7.8.1 Fragment Shader Inputs 283
7.8.2 Fragment Shader Outputs 284
7.8.3 Compiling, Linking, and Using Shaders 284
7.8.4 Setting Uniform Values 286
7.9 Basic Coloring Methods 287
7.9.1 Per-Object Colors 288
7.9.2 Per-Vertex Colors 288
7.9.3 Per-Triangle Colors 290
7.9.4 Sharp Edges and Vertex Colors 290
7.9.5 More about Basic Shading 291
7.9.6 Limitations of Basic Shading Methods 292
7.10 Texture Mapping 292
7.10.1 Introduction 292
7.10.2 Shading via Image Lookup 293
7.10.3 Texture Images 294
7.10.4 Texture Samplers 297
7.11 Texture Coordinates 297
7.11.1 Mapping Texture Coordinates onto Objects 298
7.11.2 Generating Texture Coordinates 300
7.11.3 Texture Coordinate Discontinuities 301
7.11.4 Mapping Outside the Unit Square 302
7.11.5 Texture Samplers in Shader Code 309
7.12 The Steps of Texturing 309
7.12.1 Other Forms of Texture Coordinates 310
7.12.2 From Texture Coordinates to a Texture Sample Color 311
7.13 Limitations of Static Shading 312
7.14 Chapter Summary 313

CHAPTER 8
LIGHTING **315**

8.1 Introduction 315
8.2 Basics of Light Approximation 316
8.2.1 Measuring Light 317
8.2.2 Light as a Ray 318
8.3 A Simple Approximation of Lighting 318

8.4 Types of Light Sources 319
8.4.1 Directional Lights 320
8.4.2 Point Lights 321
8.4.3 Spotlights 327
8.4.4 Other Types of Light Sources 330
8.5 Surface Materials and Light Interaction 331
8.6 Categories of Light 332
8.6.1 Emission 332
8.6.2 Ambient 332
8.6.3 Diffuse 334
8.6.4 Specular 338
8.7 Combined Lighting Equation 343
8.8 Lighting and Shading 348
8.8.1 Flat-Shaded Lighting 349
8.8.2 Per-Vertex Lighting 350
8.8.3 Per-Fragment Lighting 354
8.9 Textures and Lighting 358
8.9.1 Basic Modulation 359
8.9.2 Specular Lighting and Textures 360
8.9.3 Textures as Materials 362
8.10 Advanced Lighting 363
8.10.1 Normal Mapping 363
8.11 Reflective Objects 366
8.12 Shadows 367
8.13 Chapter Summary 368

CHAPTER 9
RASTERIZATION 369

9.1 Introduction 369
9.2 Displays and Framebuffers 370
9.3 Conceptual Rasterization Pipeline 371
9.3.1 Rasterization Stages 372
9.4 Determining the Fragments: Pixels Covered by a Triangle 373
9.4.1 Fragments 373
9.4.2 Depth Complexity 373
9.4.3 Converting Triangles to Fragments 375
9.4.4 Handling Partial Fragments 376
9.5 Determining Visible Geometry 378
9.5.1 Depth Buffering 378
9.5.2 Depth Buffering in Practice 387
9.6 Computing Fragment Shader Inputs 388
9.6.1 Uniform Values 389
9.6.2 Per-Vertex Attributes 389

9.6.3 Interpolating Texture Coordinates 392
9.6.4 Other Sources of Texture Coordinates 394
9.7 **Evaluating the Fragment Shader 395**
9.8 **Rasterizing Textures 395**
9.8.1 Texture Coordinate Review 396
9.8.2 Mapping a Coordinate to a Texel 396
9.8.3 Mipmapping 404
9.9 **From Fragments to Pixels 415**
9.9.1 Pixel Blending 416
9.9.2 Antialiasing 420
9.9.3 Antialiasing in Practice 427
9.10 **Chapter Summary 428**

CHAPTER 10
INTERPOLATION **431**

10.1 **Introduction 431**
10.2 **Interpolation of Position 433**
10.2.1 General Definitions 433
10.2.2 Linear Interpolation 435
10.2.3 Hermite Curves 438
10.2.4 Catmull-Rom Splines 448
10.2.5 Kochanek-Bartels Splines 450
10.2.6 Bézier Curves 452
10.2.7 Other Curve Types 456
10.3 **Interpolation of Orientation 458**
10.3.1 General Discussion 458
10.3.2 Linear Interpolation 461
10.3.3 Spherical Linear Interpolation 465
10.3.4 Performance Improvements 469
10.4 **Sampling Curves 470**
10.4.1 Forward Differencing 471
10.4.2 Midpoint Subdivision 473
10.4.3 Computing Arc Length 476
10.5 **Controlling Speed along a Curve 480**
10.5.1 Moving at Constant Speed 480
10.5.2 Moving at Variable Speed 485
10.6 **Camera Control 488**
10.7 **Chapter Summary 491**

CHAPTER 11
RANDOM NUMBERS **493**

11.1 **Introduction 493**

11.2 **Probability 493**
11.2.1 Basic Probability 494
11.2.2 Random Variables 497
11.2.3 Mean and Standard Deviation 501
11.2.4 Special Probability Distributions 502
11.3 **Determining Randomness 505**
11.3.1 Chi-Square Test 506
11.3.2 Spectral Test 512
11.4 **Random Number Generators 513**
11.4.1 Linear Congruential Methods 516
11.4.2 Lagged Fibonacci Methods 520
11.4.3 Carry Methods 521
11.4.4 Mersenne Twister 523
11.4.5 Conclusions 526
11.5 **Special Applications 527**
11.5.1 Integers and Ranges of Integers 527
11.5.2 Floating-Point Numbers 528
11.5.3 Nonuniform Distributions 528
11.5.4 Spherical Sampling 530
11.5.5 Disc Sampling 532
11.5.6 Noise and Turbulence 534
11.6 **Chapter Summary 538**

CHAPTER 12
INTERSECTION TESTING 541

12.1 **Introduction 541**
12.2 **Closest Point and Distance Tests 542**
12.2.1 Closest Point on Line to Point 542
12.2.2 Line–Point Distance 544
12.2.3 Closest Point on Line Segment to Point 545
12.2.4 Line Segment–Point Distance 546
12.2.5 Closest Points Between Two Lines 548
12.2.6 Line–Line Distance 550
12.2.7 Closest Points Between Two Line Segments 551
12.2.8 Line Segment–Line Segment Distance 553
12.2.9 General Linear Components 554
12.3 **Object Intersection 554**
12.3.1 Spheres 556
12.3.2 Axis-Aligned Bounding Boxes 563
12.3.3 Swept Spheres 571
12.3.4 Object-Oriented Boxes 576
12.3.5 Triangles 583

12.4 **A Simple Collision System 588**
12.4.1 Choosing a Base Primitive 589
12.4.2 Bounding Hierarchies 590
12.4.3 Dynamic Objects 591
12.4.4 Performance Improvements 593
12.4.5 Related Systems 596
12.4.6 Section Summary 599
12.5 **Chapter Summary 599**

CHAPTER 13
RIGID BODY DYNAMICS **601**

13.1 **Introduction 601**
13.2 **Linear Dynamics 602**
13.2.1 Moving with Constant Acceleration 602
13.2.2 Forces 605
13.2.3 Linear Momentum 606
13.2.4 Moving with Variable Acceleration 607
13.3 **Numerical Integration 609**
13.3.1 Definition 609
13.3.2 Euler's Method 611
13.3.3 Runge-Kutta Methods 614
13.3.4 Verlet Integration 616
13.3.5 Implicit Methods 619
13.3.6 Semi-Implicit Methods 621
13.4 **Rotational Dynamics 622**
13.4.1 Definition 622
13.4.2 Orientation and Angular Velocity 622
13.4.3 Torque 625
13.4.4 Angular Momentum and Inertia Tensor 626
13.4.5 Integrating Rotational Quantities 628
13.5 **Collision Response 630**
13.5.1 Contact Generation 630
13.5.2 Linear Collision Response 634
13.5.3 Rotational Collision Response 638
13.5.4 Extending the System 640
13.6 **Efficiency 643**
13.7 **Chapter Summary 645**

BIBLIOGRAPHY **647**
INDEX **655**
TRADEMARKS **671**
ABOUT THE CD-ROM **672**

Preface

Writing a book is an adventure. To begin with, it is a toy and an amusement; then it becomes a mistress, and then it becomes a master, and then a tyrant. The last phase is that just as you are about to be reconciled to your servitude, you kill the monster, and fling him out to the public. —Sir Winston Churchill

The Adventure Begins

As humorous as Churchill's statement is, there is a certain amount of truth to it; writing this book was indeed an adventure. There is something about the process of writing, particularly a nonfiction work like this, that forces you to test and expand the limits of your knowledge. We hope that you, the reader, benefit from our hard work.

How does a book like this come about? Many of Churchill's books began with his experience — particularly his experience as a world leader in wartime. This book had a more mundane beginning: Two engineers at Red Storm Entertainment, separately, asked Jim to teach them about vectors. These engineers were 2D game programmers, and 3D was not new, but was starting to replace 2D at that point. Jim's project was in a crunch period, so he didn't have time to do much about it until proposals were requested for the annual Game Developers Conference. Remembering the engineers' request, he thought back to the classic "Math for SIGGRAPH" course from SIGGRAPH 1989, which he had attended and enjoyed. Jim figured that a similar course, at that time titled "Math for Game Programmers," could help 2D programmers become 3D programmers.

The course was accepted, and together with a co-speaker, Marcus Nordenstam, Jim presented it at GDC 2000. The following years (2001–2002) Jim taught the course alone, as Marcus had moved from the game industry to the film industry. The subject matter changed slightly as well, adding more advanced material such as curves, collision detection, and basic physical simulation.

It was in 2002 that the seeds of what you hold in your hand were truly planted. At GDC 2002, another GDC speaker, whose name, alas, is lost to time, recommended that Jim turn his course into a book. This was an interesting idea, but how to get it published? As it happened, Jim ran into Dave Eberly at SIGGRAPH 2002, and he was looking for someone to write just that book for Morgan Kaufmann. At the same time, Lars, who was working at Numeric Design Limited at the time, was presenting some

of the basics of rendering on handheld devices as part of a SIGGRAPH course. Jim and Lars discussed the fact that handheld 3D rendering had brought back some of the "lost arts" of 3D programming, and that this might be included in a book on mathematics for game programming.

Thus, a co-authorship was formed. Lars joined Jim in teaching the GDC 2003 version of what was now called "Essential Math for Game Programmers," and simultaneously joined Jim to help with the book, helping to expand the topics covered to include numerical representations. As we began to flesh out the latter chapters of the outline, Lars was finding that the advent of programmable shaders on consumer 3D hardware was bringing more and more low-level lighting, shading, and texturing questions into his office at NDL. Accordingly, the planned single chapter on "texturing and antialiasing" became three, covering a wider selection of these rendering topics.

By early 2003, we were furiously typing the first full draft of the first edition of this book, and by GDC 2004 the book was out. Having defeated the dragon, we retired to our homes to live out the rest of our quiet little lives.

Or so we thought.

The Adventure Continues

Response to the first edition was quite positive, and the book continued to sell well beyond the initial release. Naturally, thoughts turned to what we could do to improve the book beyond what we already created.

In reviewing the topic list, it was obvious what the most necessary change was. Within a year or so of the publication of the first edition, programmable shading had revolutionized the creation of 3D applications on game consoles and on PC. While the first edition had provided readers with many of the fundamentals behind the mathematics used in shaders, it stopped short of actually discussing them in detail. It was clear that the second edition needed to embrace shaders completely, applying the mathematics of the earlier chapters to an entirely new set of rendering content. So the single biggest change in the second edition is a move to a purely shader-based rendering pipeline.

We also sent the book to reviewers to ask them what they would like to see added. The two most common requests were information about random numbers and the addition of problems and exercises. So we are providing both. A brand new chapter on probability and random numbers has been added, and problems and exercises for each chapter have been added to the CD in the back of the book. In addition, the entire book has been revised to add corrections and make the content flow better. We hope you'll find our efforts worthwhile.

Both times, the experience was fascinating, sometimes frustrating, but ultimately deeply rewarding. Hopefully, this fascination and respect for the material will be conveyed to you, the reader. The topics in this book can each take a lifetime to study to a truly great depth; we hope you will be convinced to try just that, nonetheless!

Enjoy as you do so, as one of the few things more rewarding than programming and seeing a correctly animated, simulated, and rendered scene on a screen is the confidence of understanding *how* and *why* everything worked. When something in a 3D

system goes wrong (and it *always* does), the best programmers are never satisfied with "I fixed it, but I'm not sure how;" without understanding, there can be no confidence in the solution, and nothing new is learned. Such programmers are driven by the desire to understand what went wrong, how to fix it, and learning from the experience. No other tool in 3D programming is quite as important to this process than the mathematical bases[1] behind it.

Those Who Helped Us Along the Road

In a traditional adventure the protagonists are assisted by various characters that pass in and out of the pages. Similarly, while this book bears the names of two people on the cover, the material between its covers bears the mark of many, many more. We would like to thank a few of them here.

The folks at our publisher, Elsevier, were extremely patient with both of us as we made up for being more experienced this time around by being more busy and less responsive! Chris Simpson, Laura Lewin, Georgia Kennedy, and Paul Gottehrer were all patient, professional, and flexible when we most needed it.

In addition, credit is still due to the folks at Morgan Kaufmann who helped us publish the first edition. Tim Cox, our editor, and Stacie Pierce and Richard Camp, his assistants, as well as Troy Lilly (in production) were patient and helpful in the daunting task of leading two first-time authors through the process. Special thanks are due to Dave Eberly, the series editor of our first edition, who read most of the book several times and provided great encouragement (and the occasional scolding) through the entire process, one he's been through firsthand several times.

Our reviewers were top-notch. Together, Erin Catto and Chad Robertson reviewed the entire second edition of the book. Robert Brown, Matthew McCallus, Greg Stelmack, and Melinda Theilbar were invaluable for their comments on the random numbers chapter. Ian Ashdown, Steven Woodcock, John O'Brien, J.R. Parker, Neil Kirby, John Funge, Michael van Lent, Peter Norvig, Tomas Akenine-Möller, Wes Hunt, Peter Lipson, Jon McAllister, Travis Young, Clark Gibson, Joe Sauder, and Chris Stoy each reviewed parts of the first edition or the proposals for them. Despite having tight deadlines, they all provided page after page of useful feedback, keeping us honest and helping us generate a better arc to the material. Several of them went *well* above and beyond the call of duty, providing detailed comments and even re-reading sections of the book that required significant changes. Finally, thanks also to Victor Brueggemann and Garner Halloran, who asked Jim the questions that started this whole thing off five years ago.

Jim and Lars would like to acknowledge the folks at their jobs at NVIDIA Corporation, who were *very* understanding with respect to the time-consuming process of creating a book. Also, thanks to the talented engineers at this and previous companies who provided the probing discussions and great questions that led to and continually fed this book.

1. Vector or otherwise.

In addition, Jim would like to thank Mur and Fiona, his wife and daughter, who were willing to put up with this a second time after his long absences the first time through; his sister, Liz, who provided illustrations for an early draft of this text; and his parents, Jim and Pat, who gave him the resources to make it in the world and introduced him to the world of computers so long ago.

Lars would like to thank Jen, his wife, who somehow had the courage to survive a second edition of the book even after being promised that the first edition "was it;" and his parents, Steve and Helene, who supported, nutured, and taught him so much about the value of constant learning and steadfast love.

And lastly, we would once again like to thank you, the reader, for joining us on this adventure. May the teeth of this monster find fertile ground in your minds, and yield a new army of 3D programmers.

Introduction

The (Continued) Rise of 3D Games

Over the past decade or so (driven by increasingly powerful computer and video game console hardware), three-dimensional (3D) games have expanded from custom-hardware arcade machines to the realm of hardcore PC games, to consumer set-top video game consoles, and even to handheld devices such as personal digital assistants (PDAs) and cellular telephones. This explosion in popularity has lead to a corresponding need for programmers with the ability to program these games. As a result, programmers are entering the field of 3D games and graphics by teaching themselves the basics, rather than a classic college-level graphics and mathematics education. At the same time, many college students are looking to move directly from school into the industry. These different groups of programmers each have their own set of skills and needs in order to make the transition. While every programmer's situation is different, we describe here some of the more common situations.

Many existing, self-taught 3D game programmers have strong game experience and an excellent practical approach to programming, stressing visual results and strong optimization skills that can be lacking in college-level computer science programs. However, these programmers are sometimes less comfortable with the conceptual mathematics that form the underlying basis of 3D graphics and games. This can make developing, debugging, and optimizing these systems more of a trial-and-error exercise than would be desired.

Programmers who are already established in other specializations in the game industry, such as networking or user interfaces, are now finding that they want to expand their abilities into core 3D programming. While having experience with a wide range of game concepts, these programmers often need to learn or refresh the basic mathematics behind 3D games before continuing on to learn the applications of the principles of rendering and animation.

On the other hand, college students entering (or hoping to enter) the 3D games industry often ask what material they need to know in order to be prepared to work on these games. Younger students often ask what courses they should attend in order to gain the most useful background for a programmer in the industry. Recent graduates, on the other hand, often ask how their computer graphics knowledge best relates to the way games are developed for today's computers and game consoles.

We have designed this book to provide something for each of these groups of readers. We attempt to provide readers with a conceptual understanding of the

mathematics needed to create 3D games, as well as an understanding of how these mathematical bases actually *apply* to games and graphics. The book provides not only theoretical mathematical background, but also many examples of how these concepts are used to affect how a game looks (how it is *rendered*) and plays (how objects move and react to users). Each type of reader is likely to find sections of the book that, for them, provide mainly refresher courses, a new understanding of the applications of basic mathematical concepts, or even completely new information. The specific sections that fall into each category for a particular reader will, of course, depend on the reader.

How to Read This Book

Perhaps the best way to discuss any reader's approach to reading this book is to think in terms of how a 3D game or other interactive application works at the highest level. Most readers of this book likely intend to apply what they learn from it to create, extend, or fix a 3D game or other 3D application. Each chapter in this book deals with a different topic that has applicability to some or all of the major parts of a 3D game.

Game Engines

An interactive 3D application such as a game requires quite a large amount of code to do all of the many things asked of it. This includes representing the virtual world, animating parts of it, drawing that virtual world, and dealing with user interaction in a game-relevant manner. The bulk of the code required to implement these features is generally known as a *game engine*. Game engines have evolved from small, simple, low-level rendering systems in the early 1990s to massive and complex software systems in modern games, capable of rendering detailed and expansive worlds, animating realistic characters, and simulating complex physics. At their core, these game engines are really implementations of the concepts discussed throughout this book.

Initially, game engines were custom affairs, written for a single use as a part of the game itself, and thrown away after being used for that single game project. Today, game developers have several options when considering an engine. They may purchase a commercial engine from another company and use it unmodified for their project. They may purchase an engine and modify it very heavily to customize their application. Finally, they may write their own, although most programmers choose to use such an internally developed engine for multiple games to offset the large cost of creating the engine.

In any of these cases, the developer must still understand the basic concepts of the game engine. Whether as a user, a modifier, or an author of a game engine, the developer must understand at least a majority of the concepts presented in this book. To better understand how the various chapters in this book surface in game engines, we first present a common *main loop* as it might appear in a game engine:

1. Draw the current configuration of the game's scene to the screen.
2. Animate the characters in the scene based on animator-created sequences (e.g., soccer players running downfield).

3. Detect collisions between the characters and objects (e.g., the soccer ball entering the goal or two players sliding into one another).
4. React to these collisions and basic forces such as gravity in the scene in a physically correct manner (e.g., the soccer ball in flight).

All of these steps will need to be done for each frame to present the player with a convincing game experience. Thus, the code to implement the steps above must be correct and optimal.

Chapters 1–5: The Basics

Perhaps the most core parts of any game engine are the low-level mathematical and geometric representations and algorithms. The pieces of code will be used by each and every step listed above. Chapter 1 provides the lowest-level basis for this. It discusses the practicalities of representing real numbers on a computer, with a focus on the issues most likely to affect the development of a 3D game engine for a PC, console, or handheld device.

Chapter 2 provides a focused review of vectors and points, objects that are used in all game engines to represent locations, directions, velocities, and other geometric quantities in all aspects of a 3D application. Chapters 3 and 4 review the basics of linear and affine algebra as they relate to orienting, moving, and distorting the objects and spaces that make up a virtual world. Finally, Chapter 5 introduces the quaternion, a very powerful nonmatrix representation of object orientation that will be pivotal to the later chapters on animation and simulation.

Three-dimensional engine code that implements all of these fundamental objects must be built carefully and with a good understanding of both the underlying mathematics and programming issues. Otherwise, the game engine built on top of these basic objects or functions will be based upon a poor foundation. Many game programmers' multiday debugging sessions have ended with the realization that the complex bug was rooted in an error in the engine's basic mathematics code.

Some readers will have a passing familiarity with the topics in these chapters. However, most readers will want to start with these chapters, as many of the topics are covered in more conceptual detail than is often discussed in basic graphics texts. Readers new to the material will want to read in detail, while those who already know some linear algebra can use the chapters to fill in any missing background. All of these chapters form a basis for the rest of the book, and an understanding of these topics, whether existing or new, will be key to successful 3D programming.

Chapters 6–9: Rendering

Chapters 6–9 apply the foundational objects detailed in Chapters 1–5 to explain step 1 of the game engine main loop: the rendering or drawing pipeline, perhaps the best-known part of any game engine. In some game engines, more time and effort is spent designing, programming, and tuning the rendering pipeline than the rest of the engine in its entirety. Chapter 6 describes the mathematics and geometry behind the virtual cameras used to view the scene or game world. Chapter 7 describes the representation

of color and the concept of *shaders*, which are short programs that allow modern graphics hardware to draw the scene objects to the display device. Chapter 8 explains how to use these programmable shaders to implement simple approximations of real-world lighting. The rendering section concludes with Chapter 9, which details the methods used by low-level rendering systems to draw to the screen. An understanding of these details can allow programmers to create much more efficient and artifact-free rendering code for their engines.

Chapters 10–13: Animation and Physics

The game engine loop's step 2, animating characters and other objects based on data created by computer animators or motion-captured data, is introduced in Chapter 10. This chapter discusses methods for smoothly animating the position, orientation, and appearance of objects in the virtual game world. The importance of good, complex character and object animation in modern engines continues to grow as new games attempt to create smoother, more convincing representations of athletes, rock stars, soldiers, and other human characters.

Chapter 11 covers another element for adding realism to games: random numbers. Everything up to this point has been carefully determined and planned by the programmer or artist. Adding randomness adds the unexpected behavior that we see in real life. Gunshots are not always exact, clouds are not perfectly spherical, and walls are not pristine. This chapter discusses how to handle randomness in a game, and how we can get effects such as those discussed above.

Step 3, detecting collisions, is discussed in Chapter 12. This chapter describes the mathematics and programming behind detecting when two game objects touch, intersect, or penetrate. Many genres of game have exacting requirements when it comes to collision, be it a racing game, a sports title, or a military simulation.

Finally, step 4, reacting in a realistic manner to physical forces and collisions, is covered in Chapter 13. This chapter describes how to make game objects behave and react in physically convincing ways.

Put together, the chapters that form this book are designed to give a good basis for the foundations of a game engine, details on the ways that engines represent and draw their virtual worlds, and an introduction to making those worlds seem real and active.

Interactive Demo Applications

Three-dimensional games and graphics are, by their nature, not only visual but dynamic. While figures are indeed a welcome necessity in a book about 3D applications, interactive demos can be even more important. It is difficult to truly understand such topics as lighting, quaternion interpolation, or physical simulation without being able to see them work firsthand and to interact with these complex systems. This book includes a CD-ROM of source code and demonstrations that are designed

to illustrate the concepts in a way that is analogous to the static figures in the book itself. Throughout the book, you will find references to interactive demos that may be found on the CD-ROM. Whenever a topic is illustrated with an interactive demo, a special icon like the one seen next to this paragraph will appear in the margin.

Support Libraries

In addition to the source code for each of the demos, the CD-ROM includes the supporting libraries used to create the demos, with full source code. Often, code from these supporting libraries is excerpted in the book itself in order to explain how the particular concept is implemented. In such situations, an icon will appear in the margin to note where the library code may be found on the CD-ROM. This source code is designed to allow readers to modify and experiment themselves, as a way of better understanding the way the code works.

The source code is written entirely in C++, a language that is likely to be familiar to most game developers. C++ was chosen because it is one of the most commonly used languages in 3D game development and because vectors, matrices, quaternions, and graphics algorithms decompose very well into C++ classes. In addition, C++'s support of operator overloading means that the math library can be implemented in a way that makes the code look very similar to the mathematical derivations in the text. However, in some sections of the text, the class declarations as printed in the book are not complete with respect to the code on the CD-ROM. Often, class members that are not relevant to the particular discussion (especially member variable accessor and "housekeeping" functions) have been omitted for clarity. These other functions may be found in the full class declarations/definitions on the CD-ROM.

Note that we have modified our mathematical notation slightly to allow our equations to be as compatible as possible with the code. Mathematicians normally start indexing with 1, for example, $P_1, P_2, \ldots, P_n$. This does not match how indexing is done in C++: `P[0]` is the first element in the array `P`. To avoid this disconnect, in our equations we will be using the convention that the starting element in a list is indexed as 0; thus, $P_0, P_1, \ldots, P_{n-1}$. This should allow for a direct translation from equation to code.

Math Libraries

All of the demos use a shared core math library called `IvMath`, which includes C++ classes that implement vectors and matrices of different dimensions, along with a few other basic mathematical objects discussed in the book. This library is designed to be useful to readers beyond the examples supplied with the book, as the library includes a wide range of functions and operators for each of these objects, some of which are beyond the scope of the book's demos.

The animation demos use a shared library called `IvCurves`, which includes classes that implement spline curves, the basic objects used to animate position, `IvCurves` is built upon `IvMath`, extending this basic functionality to include animation. As with `IvMath`, the `IvCurves` library is likely to be useful beyond the scope of the book, as these classes are flexible enough to be used (along with `IvMath`) in other applications.

Finally, the simulation demos use a shared library called `IvCollision`, which implements basic object intersection (collision) data structures and algorithms. Building on the `IvMath` library, this set of classes and functions forms not only the basis for the later demos in the book, but also is an excellent starting point for experimentation with other forms of object collision and physics modeling.

Engine and Rendering Libraries

In addition to the math libraries, the CD-ROM includes a set of classes that implement a simple game like application framework, basic rendering, input handling, and timer functionality. All of these functions are grouped under the heading of game engine functionality, and are located in the `IvEngine` library. The engine's rendering code takes the form of a set of renderer-abstraction classes that simplify the interfaces between the C++ classes in `IvMath` and the C-based, low-level rendering application programmer interfaces (APIs). This code is included as a part of the rendering library `IvGraphics`. It includes renderer setup, basic render-state management, and rendering of simple geometric primitives, such as spheres, cubes, and boxes.

Furthermore, a set of basic classes that implement a simple hierarchial data structure called a scene graph are included in the library `IvScene`. The classes in `IvScene` use and depend on the functionality of the `IvCollision` library. As a result, to avoid unnecessary code dependencies, the scene graph classes were placed in their own library, rather than in `IvEngine`.

Since this book focuses on the mathematics and concepts behind 3D games, we chose not to center the discussion around a large-scale, general 3D rendering engine. Doing so would introduce an extra layer of indirection that would not serve the conceptual requirements of the book. Valuable real estate in the rendering chapters would be spent on background in the use of a particular engine — the one written for the book. For an example and discussion of a full, hierarchical rendering engine, the reader is encouraged to read David Eberly's *3D Game Engine Design* [25].

We have opted to implement our rendering system and examples using two standard SDKs: the multiplatform OpenGL [83] and the popular Direct3D DX9 [47]. We also use the utility toolkits provided with these SDKs (OpenGL's GLUT and Direct3D's D3DX) to implement cross-platform renderer setup and input handling, neither of which are core topics of this book.

Exercises and Supplementary Material

In addition to the sample code, we have included some useful reading material on the CD-ROM for those who haven't absorbed enough of our luminous prose. Each chapter

has an associated set of exercises, ranging from easy to hard questions, that should help those readers interested in testing their understanding of the material within. Certain chapters also have supplemental material that unfortunately didn't make its way into the book proper due to space considerations. Those chapters have notes at their end indicating that such material is available on the CD-ROM.

References and Further Reading

Hopefully, this book will leave readers with a desire to learn even more details and the breadth of the mathematics involved in creating high-performance, high-quality 3D games. Wherever possible, we have included references to other books, articles, papers, and websites that detail particular subtopics that fall outside the scope of this book. The full set of references may be found at the back of the book.

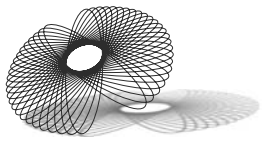
We have attempted to include references that the vast majority of readers should be able to locate. When possible, we have referenced recent and/or standard industry texts and well-known conference proceedings. However, in some cases we have included references to older magazine articles and technical reports when we found those references to be particularly complete, seminal, or well written. In some cases, older references can be easier for the less-experienced reader to understand, as they often tend to assume less common knowledge when it comes to computer graphics and game topics.

In the past, older magazine articles and technical reports were notoriously difficult for the average reader to locate. However, the Internet and digital publishing have made great strides toward reversing this trend. For example, the following sources have made several classes of resources far more accessible:

- The magazine most commonly referenced in this book, *Game Developer*, offers CD-ROMs that contain every issue of the magazine ever published. Copies of these CD-ROMs are available from *www.gdmag.com*. Several other technical magazines also offer such CD-ROMs.
- Technical societies are now placing major historical publications into their "digital libraries," which are often made accessible to members. The Association for Computing Machinery (ACM) has done this via their ACM Digital Library, which is available to ACM members. As an example, the full text of the entire collection of papers from all SIGGRAPH conferences (the conference proceedings most frequently referenced in this book) is available electronically to ACM SIGGRAPH members.
- Other papers and technical reports are often available on the Internet. The two most common methods of finding these resources are via publication portals such as Citeseer (*www.citeseer.com*) and via the authors' personal homepages (if they have them). Most of the technical reports referenced in this book are available online from such sources. Owing to the dynamic nature of the Internet, we suggest using a search engine if the publication portals do not succeed in finding the desired article.

For further reading, we suggest several books that cover topics related to this book in much greater detail. In most cases they assume that the reader is familiar with the concepts discussed in this book. David Eberly's *3D Game Engine Design* [25] discusses the design and implementation of a full game engine, focusing mostly on graphics and animation. Books by Gino van den Bergen [108] and Christer Ericson [32] cover topics in interactive collision detection. Finally, Eberly [27] and Millington [76] provide a more advanced discussion of a wide range of physical simulation topics.

Chapter 1



Real-World Computer Number Representation

1.1 Introduction

In this chapter we'll discuss what is perhaps the most fundamental basis upon which three-dimensional (3D) graphics pipelines are built: computer representation of numbers, particularly real numbers. While 3D programmers often use the computer representations (approximations) of real numbers successfully without any understanding of how they are implemented, this can lead to subtle bugs and performance problems at inopportune stages in the development of an application. Most basic undergraduate computer architecture books [106] present the basics of integral data types (e.g., `int` and `unsigned int`, `short`, etc. in C/C++), but give only brief introductions to floating-point and other nonintegral number representations. Since the mathematics of 3D graphics are generally real-valued (as we shall see from the ubiquity of $\mathbb{R}$, $\mathbb{R}^2$, and $\mathbb{R}^3$ in the coming chapters), it is important for anyone in the field to understand the features, limitations, and idiosyncracies of the computer representation of these nonintegral types.

In this chapter we will discuss the major computer representation of the real numbers, floating-point, along with the associated bitwise formats, basic operations, features, and limitations. By design, we will transition from general mathematical discussions of number representation toward implementation-related topics of specific relevance to 3D graphics programmers. Most of the chapter will be spent on the ubiquitous Institute of Electrical and Electronic Engineers (IEEE) floating-point numbers, especially discussions of floating-point limitations that often cause issues in

3D pipelines. A brief case study of floating-point-related performance issues in a real application is also presented.

We will assume that the reader is familiar with the basic concepts of integer and whole-number representations on modern computers, including signed representation via two's complement, range, overflow, common storage lengths (8,16, and 32 bits), standard C and C++ basic types (`int`, `unsigned int`, `short`, etc.), and type conversion. For an introduction to these concepts of binary number representation, we refer the reader to a basic computer architecture text, such as Stallings [106], and to the C++ specification [30].

1.2 Representing Real Numbers

Real numbers are, to most developers, the heart and soul of a 3D graphics system. Most of the rest of the text is based upon real numbers and spaces such as $\mathbb{R}^2$ and $\mathbb{R}^3$. They are the most flexible and powerful of the number representations on most computers and, not surprisingly, the most complicated and problematic. We will present the methods that are used to represent real numbers on computers today and will include numerous sections describing common issues that arise from the use of these representations in real-world applications.

The well-known issues relating to storage of integers (such as overflow) remain pertinent issues with respect to real-number representation. However, real-number representations add additional complexities that will result in implementation trade-offs, subtle errors, and difficult-to-trace performance issues that can easily confuse the programmer.

1.2.1 Approximations

While computer representations of whole numbers (`unsigned int`) and integers (`int`) are limited to a finite subset of their pure counterparts, in each case the finite set is contiguous; that is, if i and $i + 2$ are both representable, then $i + 1$ is also representable. Inside the range defined by the minimum and maximum representable integer values, all integers can be represented exactly. This is possible because any finitely bounded range of integers contains a finite number of elements.

When dealing with real numbers, however, this is no longer true. A subset of real numbers can have infinitely many elements even when bounded by finite minimal and maximal values. As a result, no matter how tightly we bound the range of real numbers (other than the trivial case of $R_{min} = R_{max}$) that we choose to represent, we will be unable to represent that subset of the real numbers exactly. Issues of both range *and* precision will thus be constant companions over the course of our discussion of real-number representations.

In order to adequately understand the representations of real numbers, we need to understand the concept of precision and error.

1.2.2 Precision and Error

For any numerical representation system, we imagine a generic function $Rep(A)$, which returns the value in that system that is closest to the value A. In a perfect representation system, $Rep(A) = A$ for all values of A. When representing real numbers on a computer, however, even limiting range to finite extremes will not allow us to represent all numbers in the bounded range exactly. $Rep(A)$ will be a many-to-one mapping, with infinitely many real numbers A mapping to each distinct value returned by $Rep(A)$. For each such distinct $Rep(A)$, almost all values A that map to it will not be represented exactly. In other words, for almost all real values A, $Rep(A) \neq A$. The obvious result in such cases is that $(Rep(A) - A) \neq 0$. The representation in such a case is an approximation of the actual value.

Making use of $(Rep(A) - A)$, we can define several derived values that form metrics of the error induced by representing A in the representation system. Two such error metrics are called *absolute error* and *relative error*.

The simplest way to represent error is absolute error, which is defined as

$$AbsError = |Rep(A) - A|$$

This is simply the "number line" distance between the actual value and its representation. While this value does correctly signify the difference between the actual and representative values, it does not quantify another important factor in representation error — the scale at which the error affects computation.

To better understand this scale factor, imagine a system of measurement that is accurate to within a kilometer. Such a system might be considered suitably accurate for measuring the 149,597,871 km between Earth and the sun. However, it likely would be woefully inaccurate at measuring the size of an apple (0.00011 km), which would be rounded to 0 km! Intuitively, this is obvious, but in both cases the absolute error of representation is less than 1 km. Clearly, absolute error is not sufficient in all cases.

Relative error takes the scale of the value being approximated into account. It does so by dividing the absolute error by the actual value being represented. Relative error is defined as

$$RelError = \left| \frac{Rep(A) - A}{A} \right|$$

As such, relative error is dimensionless; even if the values being approximated have units (such as kilometers), the relative error has no units. Due to the

division, relative error cannot be computed for a value that approximates zero. It is a measure of the ratio of the error to the magnitude of the value being approximated. Revisiting our previous example, the relative errors in each case would be (approximately)

$$RelError_{Sun} = \left| \frac{1\,\text{km}}{149{,}597{,}871\,\text{km}} \right| \approx 7 \times 10^{-9}$$

$$RelError_{Apple} = \left| \frac{0.00011\,\text{km}}{0.00011\,\text{km}} \right| = 1.0$$

Clearly, relative error is a much more useful error metric in this case. The Earth–sun distance error is tiny (compared to the distance being measured), while the size of the apple was estimated so poorly that the error had the same magnitude as the actual value. In the former case a relatively "exact" representation was found, while in the latter case the representation is all but useless.

1.3 Floating-Point Numbers

1.3.1 Review: Scientific Notation

In order to better introduce floating-point numbers, it is instructive to review the well-known standard representation for real numbers in science and engineering: scientific notation. Computer floating-point is very much analogous to scientific notation.

Scientific notation (in its strictest, so-called normalized form) consists of two parts:

1. A decimal number, called the *mantissa*, such that

$$1.0 \leq |mantissa| < 10.0$$

2. An integer, called the *exponent*.

Together, the exponent and mantissa are combined to create the number

$$mantissa \times 10^{exponent}$$

Any decimal number can be represented in this notation (other than 0, which is simply represented as 0.0), and the representation is unique for each number. In other words, for two numbers written in this form of scientific notation, the numbers are equal if and only if their mantissas and exponents are equal. This uniqueness is a result of the requirements that the exponent be an integer and that the mantissa be "normalized" (i.e., have magnitude in the range [1.0, 10.0]). Examples of numbers written in scientific notation include

$$\begin{aligned} 102 &= 1.02 \times 10^2 \\ 243{,}000 &= 2.43 \times 10^5 \\ -0.0034 &= -3.4 \times 10^{-3} \end{aligned}$$

Examples of numbers that constitute incorrect scientific notation include

$$\begin{aligned} \text{Incorrect} &= \text{Correct} \\ 11.02 \times 10^3 &= 1.102 \times 10^4 \\ 0.92 \times 10^{-2} &= 9.2 \times 10^{-3} \end{aligned}$$

1.3.2 A Restricted Scientific Notation

For the purpose of introducing the concept of finiteness of representation, we will briefly discuss a contrived, restricted scientific notation. We extend the rules for scientific notation:

1. The mantissa must be written with a single, nonzero integral digit.
2. The mantissa must be written with a fixed number of fractional digits (we define this as M digits).
3. The exponent must be written with a fixed number of digits (we define this as E digits).
4. The mantissa and the exponent each have individual signs.

For example, the following number is in a format with $M = 3$, $E = 2$:

$$\pm\boxed{1.1}\boxed{2}\boxed{3} \times 10^{\pm\boxed{1}\boxed{2}}$$

Limiting the number of digits allocated to the mantissa and exponent means that any value that can be represented by this system can be represented uniquely by six decimal digits and two signs. However, this also implies that

there are a limited number of values that could ever be represented exactly by this system, namely:

$$\begin{aligned}&(\text{exponents}) \times (\text{mantissas}) \times (\text{exponent signs}) \times (\text{mantissa signs})\\ &= (10^2) \times (9 \times 10^3) \times (2) \times (2)\\ &= 3{,}600{,}000\end{aligned}$$

Note that the leading digit of the mantissa must be nonzero (since the mantissa is normalized), so that there are only nine choices for its value [1, 9], leading to $9 \times 10 \times 10 \times 10 = 9{,}000$ possible mantissas.

This adds finiteness to both the range and precision of the notation. The minimum and maximum exponents are

$$\pm(10^E - 1) = \pm(10^2 - 1) = \pm 99$$

The largest mantissa value is

$$10.0 - (10^{-M}) = 10.0 - (10^{-3}) = 10.0 - 0.001 = 9.999$$

Note that the smallest allowed nonzero mantissa value is still 1.000 due to the requirement for normalization. This format has the following numerical limitations:

$$\begin{aligned}\text{Maximum representable value: } & 9.999 \times 10^{99}\\ \text{Minimum representable value: } & -9.999 \times 10^{99}\\ \text{Smallest positive value: } & 1.000 \times 10^{-99}\end{aligned}$$

While one would likely never use such a restricted form of scientific notation in practice, it demonstrates the basic building blocks of binary floating-point, the most commonly used computer representation of real numbers in modern computers.

1.4 Binary "Scientific Notation"

There is no reason that scientific notation must be written in base-10. In fact, in its most basic form, the real-number representation known as floating-point is similar to a base-2 version of the restricted scientific notation given previously. In base-2, our restricted scientific notation would become

$$SignM \times mantissa \times 2^{SignE \times exponent}$$

where *exponent* is an E-bit integer, and *SignM* and *SignE* are independent bits representing the signs of the mantissa and exponent, respectively.

Mantissa is a bit more complicated. It is an $M+1$-bit number whose most significant bit is 1. *Mantissa* is actually a "fixed-point" number. Fixed-point numbers are based on a very simple observation with respect to computer representation of integers. In the standard binary representation, each bit represents twice the value of the bit to its right, with the least significant bit representing 1. The following diagram shows these powers of two for a standard 8-bit unsigned value:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Just as a decimal number can have a decimal point, which represents the break between integral and fractional values, a binary value can have a binary point, or more generally a radix point (a decimal number is referred to as radix 10, a binary number as radix 2). In the common integer number layout, we can imagine the radix point being to the right of the last digit. However, it does not have to be placed there. For example, let us place the radix point in the middle of the number (between the fourth and fifth bits). The diagram would then look like this:

2^3	2^2	2^1	2^0 . 2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1 . $\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$

Now, the least significant bit represents 1/16. The basic idea behind fixed-point is one of scaling. A fixed-point value is related to an integer with the same bit pattern by an implicit scaling factor. This scaling factor is fixed for a given fixed-point format and is the value of the least significant bit in the representation. In the case of the preceding format, the scaling factor is 1/16.

The standard nomenclature for a fixed-point format is "A-dot-B," where A is the number of integral bits (to the left of the radix point) and B is the number of fractional bits (to the right of the radix point). For example, the 8-bit format in our example would be referred to as "4-dot-4." As a further example, regular 32-bit integers would be referred to as "32-dot-0" because they have no fractional bits. More generally, the scaling factor for an A-dot-B format is simply 2^{-B}. Note that, as expected, the scaling factor for a 32-dot-0 format (integers) is $2^0 = 1$. No matter what the format, the radix point is "fixed" (or locked) at B bits from the least significant bit; thus the name "fixed-point."

Since the mantissa is a 1-dot-M fixed-point number, the leading bit represents the integer 1. As mentioned above, the leading bit in the mantissa is

defined to be 1, so the resulting fixed-point *mantissa* is in the range

$$1.0 \leq mantissa \leq \left(2.0 - \frac{1}{2^M}\right)$$

Put together, the format involves $M + E + 3$ bits ($M + 1$ for the mantissa, E for the exponent, and two for the signs). Creating an example that is analogous to the preceding decimal case, we analyze the case of $M = 3$, $E = 2$:

$$\pm\boxed{1.0}\boxed{1}\boxed{0} \times 2^{\pm\boxed{0}\boxed{1}}$$

Any value that can be represented by this system can be represented uniquely by 8 bits. The number of values that ever could be represented exactly by this system is

$$\begin{aligned}
&(\text{exponents}) \times (\text{mantissas}) \times (\text{exponent signs}) \times (\text{mantissa signs}) \\
&= (2^2) \times (1 \times 2^3) \times (2) \times (2) \\
&= 2^7 = 128
\end{aligned}$$

This seems odd, as an 8-bit number should have 256 different values. However, note that the leading bit of the mantissa must be 1, since the mantissa is normalized (and the only choices for a bit's value are 0 and 1). This effectively fixes one of the bits and cuts the number of possible values in half. We shall see that the most common binary floating-point format takes advantage of the fact that the integral bit of the mantissa is fixed at 1.

In this case, the minimum and maximum exponents are

$$\pm(2^E - 1) = \pm(2^2 - 1) = \pm 3$$

The largest mantissa value is

$$2.0 - 2^{-M} = 2.0 - 2^{-3} = 1.875$$

This format has the following numerical limitations:

$$\begin{aligned}
\text{Maximum representable value:} &\quad 1.875 \times 2^3 = 15 \\
\text{Minimum representable value:} &\quad -1.875 \times 2^3 = -15 \\
\text{Smallest positive value:} &\quad 1.000 \times 2^{-3} = 0.125
\end{aligned}$$

From the listed limits, it is quite clear that a floating-point format based on this simple 8-bit binary notation would not be useful to most real-world applications. However, it does introduce the basic concepts that are shared by real floating-point representations. While there are countless possible floating-point formats, the universal popularity of a single set of formats (those described in the IEEE 754 specification [2]) makes it the obvious choice for any discussion of the details of floating-point representation. In the remainder of this chapter we will explain the major concepts of floating-point representation as evidenced by the IEEE standard format.

1.5 IEEE 754 Floating-Point Standard

By the early to mid-1970s, scientists and engineers were using floating-point very frequently to represent real numbers; at the time, higher-powered computers even included special hardware to accelerate floating-point calculations. However, these same scientists and engineers were finding the lack of a floating-point standard to be problematic. Their complex (and often very important) numerical simulations were producing different results, depending only on the make and model of computer upon which the simulation was run. Numerical code that had to run on multiple platforms became riddled with platform-specific code to deal with the differences between different floating-point processors and libraries.

In order for cross-platform numerical computing to become a reality, a standard was needed. Over the course of the next decade, a draft standard for floating-point formats and behaviors became the de facto standard on most floating-point hardware. Once adopted, it became known as the IEEE 754 floating-point standard [2], and it forms the basis of almost every hardware and software floating-point system on the market.

While the history of the standard is fascinating [62], this section will focus on explaining part of the standard itself, as well as using the standard and one of its specified formats to explain the concepts of modern floating-point arithmetic.

1.5.1 Basic Representation

The IEEE standard specifies a 32-bit "single-precision" format for floating-point numbers, as well as a 64-bit "double-precision" format. It is this single-precision format that is of greatest interest for most games and interactive applications and is thus the format that will form the basis of most of the floating-point discussion in this text. The two formats are fundamentally similar, so all of the concepts regarding single precision are applicable to double-precision values as well.

The following diagram shows the basic memory layout of the IEEE single-precision format, including the location and size of the three components of any floating-point system: sign, exponent, and mantissa:

Sign	Exponent	Mantissa
1 bit	8 bits	23 bits

The sign in the IEEE floating-point format is represented as an explicit bit (the high-order bit). Note that this is the sign of the number itself (the mantissa), not the sign of the exponent. Differentiating between positive and

negative exponents is handled in the exponent itself (and is discussed next). The only difference between X and $-X$ in IEEE floating-point is the high-order bit. A sign bit of 0 indicates a positive number, and a sign bit of 1 indicates a negative number.

This sign bit format allows for some efficiencies in creating a floating-point math system either in hardware or software. To negate a floating-point number, simply "flip" the sign bit, leaving the rest of the bits unchanged. To compute the absolute value of a floating-point number, simply set the sign bit to 0 and leave the other bits unchanged. In addition, the sign bits of the result of a multiplication or division are simply the exclusive-OR of the sign bits of the operands.

As will be seen, this explicit sign bit does lead to the existence of two zero values, one positive and one negative. However, it also simplifies the representation of the mantissa, which is represented as unsigned.

The exponent in this case is stored as a biased number. Biased numbers represent both positive and negative integers (inside of a fixed range) as whole numbers by adding a fixed, positive bias. To represent an integer I, we add a positive bias B (that is constant for the biased format), storing the result as the whole number (nonnegative integer) W. To decode the represented value I from its biased representation W, the formula is simply

$$I = W - B$$

To encode an integer value, the formula is

$$W = I + B$$

Clearly, the minimum integer value that can be represented is

$$I = 0 - B = -B$$

The maximal value that can be represented is related to the maximum whole number that can be represented, W_{max}. For example, with an 8-bit biased number, that value is

$$I = W_{max} - B = (2^8 - 1) - B$$

Most frequently, the bias chosen is as close as possible to $W_{max}/2$, giving a range that is equally distributed to about zero. Over the course of this chapter, when we are referring to a biased number, the term *value* will refer to I, while the term *bits* will refer to W.

Such is the case with the IEEE floating-point exponent, which uses 8 bits of representation and a bias of 127. This would seem to lead to minimum and

maximum exponents of -127 ($= 0 - 127$) and 128 ($= 255 - 127$), respectively. However, for reasons that will be explained, the minimum and maximum values (-127 and 128) are reserved for special cases, leading to an exponent range of $[-126, 127]$. As a reference, these base-2 exponents correspond to base-10 exponents of approximately $[-37, 38]$.

The mantissa is normalized (in almost all cases), as in our discussion of decimal scientific notation (where the units digit was required to have magnitude in the range $[1, 9]$). However, the meaning of "normalized" in the context of a binary system means that the leading bit of the mantissa is always 1. Unlike a decimal digit, a binary digit has only one nonzero value. To optimize storage in the floating-point format, this leading bit is omitted, or hidden, freeing all 23 explicit mantissa bits to represent fractional values (and thus these explicit bits are often called the "fractional" mantissa bits). To decode the entire mantissa into a rational number (ignoring for the moment the exponent), assuming the fractional bits (as a 23-bit unsigned integer) are in F, the conversion is

$$1.0 + \frac{F}{2.0^{23}}$$

So, for example, the fractional mantissa bits

$$11100000000000000000000_2 = 7340032_{10}$$

become the rational number

$$1.0 + \frac{7340032.0}{2.0^{23}} = 1.875$$

1.5.2 Range and Precision

The range of single-precision floating-point is by definition symmetric, as the system uses an explicit sign bit. With an explicit sign bit, every positive value has a corresponding negative value. This leaves the questions of maximal exponent and mantissa, which when combined will represent the explicit values of greatest magnitude. In the previous section, we found that the maximum base-2 exponent in single precision floating-point is 127. The largest mantissa would be equal to setting all 23 explicit fractional mantissa bits, resulting (along with the implicit 1.0 from the hidden bit) in a mantissa of

$$1.0 + \sum_{i=1}^{23} \frac{1}{2^i} = 1.0 + 1.0 - \frac{1}{2^{23}} = 2.0 - \frac{1}{2^{23}} \approx 2.0$$

The minimum and maximum single-precision floating-point values are then

$$\pm\left(2.0 - \frac{1}{2^{23}}\right) \times 2^{127} \approx \pm 3.402823466 \times 10^{38}$$

The precision of single-precision floating-point can be loosely approximated as follows: For a given normalized mantissa, the difference between it and its nearest neighbor is 2^{-23}. To determine the actual spacing between a floating-point number and its neighbor, the exponent must be known. Given an exponent E, the difference between two neighboring single-precision values is

$$\delta_{fp} = 2^E \times 2^{-23} = 2^{E-23}$$

However, we note that in order to represent a value A in single precision, we must find the exponent E_A such that the mantissa is normalized (i.e., the mantissa M_A is in the range $1.0 \leq M_A < 2.0$), or

$$1.0 \leq \frac{|A|}{2^{E_A}} < 2.0$$

Multiplying through, we can bound $|A|$ in terms of 2^{E_A}:

$$\begin{aligned} 1.0 &\leq \frac{|A|}{2^{E_A}} < 2.0 \\ 2^{E_A} &\leq |A| < 2^{E_A} \times 2.0 \\ 2^{E_A} &\leq |A| < 2^{E_A+1} \end{aligned}$$

As a result of this bound, we can roughly approximate the entire exponent term 2^{E_A} with $|A|$ and substitute to find an approximation of the distance between neighboring floating-point values around $|A|$ (δ_{fp}) as

$$\delta_{fp} = 2^{E_A-23} = \frac{2^{E_A}}{2^{23}} \approx \frac{|A|}{2^{23}}$$

From our initial discussion of absolute error, we use general bound on the absolute error equal to half the distance between neighboring representation values:

$$AbsError_A \approx \delta_{fp} \times \frac{1}{2} = \frac{|A|}{2^{23}} \times \frac{1}{2} = \frac{|A|}{2^{24}}$$

This approximation shows that the absolute error of representation in a floating-point number is directly proportional to the magnitude of the value being represented. Having approximated the absolute error, we can approximate the relative error as

$$RelError_A = \frac{AbsError_A}{|A|} \approx \frac{|A|}{2^{24} \times |A|} = \frac{1}{2^{24}} \approx 6 \times 10^{-8}$$

The relative error of representation is thus generally constant, regardless of the magnitude of A.

1.5.3 Arithmetic Operations

In the next several sections we discuss the basic methods used to perform common arithmetic operations upon floating-point numbers. While few users of floating-point will ever need to implement these operations at a bitwise level themselves, a basic understanding of the methods is a pivotal step toward being able to understand the limitations of floating-point. The methods shown are designed for ease of understanding and do not represent the actual, optimized algorithms that are implemented in hardware.

The IEEE standard specifies that the basic floating-point operations of a compliant floating-point system must return values that are equivalent to the result computed exactly and *then* rounded to the available precision. The following sections are designed as an introduction to the basics of floating-point operations and do not discuss the exact methods used for rounding the results. At the end of the section, there is a discussion of the programmer-selectable rounding modes specified by the IEEE standard.

The intervening sections include information regarding common issues that arise from these operations, because each operation can produce problematic results in specific situations.

Addition and Subtraction

In order to add a pair of floating-point numbers, the mantissas of the two addends first must be shifted such that their radix points are "lined up." In a floating-point number, the radix points are aligned if and only if their exponents are equal. If we raise the exponent of a number by one, we must shift its mantissa to the right by 1 bit. For simplicity, we will first discuss addition of a pair of positive numbers. The standard floating-point addition method works (basically) as follows to add two positive numbers $A = S_A \times M_A \times 2^{E_A}$ and $B = S_B \times M_B \times 2^{E_B}$, where $S_A = S_B = 1.0$ due to the current assumption that A and B are nonnegative.

1. Swap A and B if needed so that $E_A \geq E_B$.
2. Shift M_B to the right by $E_A - E_B$ bits. If $E_A \neq E_B$, then this shifted M_B will not be normalized, and M_B will be less than 1.0. This is needed to align the radix points.
3. Compute M_{A+B} by adding the shifted mantissas M_A and M_B directly.
4. Set $E_{A+B} = E_A$.

5. The resulting mantissa M_{A+B} may not be normalized (it may have an integral value of 2 or 3). If this is the case, shift M_{A+B} to the right 1 bit and add 1 to E_{A+B}.

Note that there are some interesting special cases implicit in this method. For example, we are shifting the smaller number's mantissa to the right to align the radix points. If the two numbers differ in exponents by more than the number of mantissa bits, then the smaller number will have all of its mantissa shifted away, and the method will add zero to the larger value. This is important to note, as it can lead to some very strange behavior in applications. Specifically, if an application repeatedly adds a small value to an accumulator, as the accumulator grows, there will come a point at which adding the small value to the accumulator will result in no change to the accumulator's value (the delta value being added will be shifted to zero each iteration)!

Floating-point addition must take negative numbers into account as well. There are three distinct cases here:

- *Both operands positive*. Add the two mantissas as is and set the result sign to positive.
- *Both operands negative*. Add the two mantissas as is and set the result sign to negative.
- *One positive operand and one negative operand*. Negate (2's complement) the mantissa of the negative number and add.

In the case of subtraction (or addition of numbers of opposite sign), the result may have a magnitude that is significantly smaller than either of the operands, including a result of zero. If this is the case, there may be considerable shifting required to reestablish the normalization of the result, shifting the mantissa to the left (and shifting zeros into the lowest-precision bits) until the integral bit is 1. This shifting can lead to precision issues (see Section 1.5.6, Catastrophic Cancelation) and can even lead to nonzero numbers that cannot be represented by the normalized format discussed so far (see Section 1.5.5, Very Small Values).

We have purposefully omitted discussion of rounding, as rounding the result of an addition is rather complex to compute quickly. This complexity is due to the fact that one of the operands (the one with the smaller exponent) may have bits that are shifted out of the operation, but must still be considered to meet the IEEE standard of "exact result, then rounded." If the method were simply to ignore the shifted bits of the smaller operand, the result could be incorrect. You may want to refer to Hennessy and Patterson [59] for details on the floating-point addition algorithm.

Multiplication

Multiplication is actually rather straightforward with IEEE floating-point numbers. Once again, the three components that must be computed are the sign, the exponent, and the mantissa. As in the previous section, we will give the example of multiplying two floating-point numbers, A and B.

Owing to the fact that an explicit sign bit is used, the sign of the result may be computed simply by computing the exclusive-OR of the sign bits, producing a positive result if the signs are equal and a negative result otherwise. The result of the multiplication algorithm is sign-invariant.

To compute the initial exponent (this initial estimate may need to be adjusted at the end of the method if the initial mantissa of the result is not normalized), we simply sum the exponents. However, since both E_A and E_B contain a bias value of 127, the sum will contain a bias of 254. We must subtract 127 from the result to reestablish the correct bias:

$$E_{A\times B} = E_A + E_B - 127$$

To compute the result's mantissa, we multiply the normalized source mantissas M_A and M_B as 1-dot-23 format fixed-point numbers. The method for multiplying two X-dot-Y bit-format fixed-point numbers is to multiply them using the standard integer multiplication method and then divide the result by 2^Y (which can be done by shifting the result to the right by Y bits). For 1-dot-23 format source operands, this produces a (possibly unnormalized) 3-dot-46 result. Note from the format that the number of integral bits may be 3, as the resulting mantissa could be rounded up to 4.0. Since the source mantissas are normalized, the resulting mantissa (if it is not 0) must be ≥ 1.0, leading to three possibilities for the mantissa $M_{A\times B}$: it may be normalized, it may be too large by 1 bit, or it may be too large by 2 bits. In the latter two cases, we add either 1 or 2 to $E_{A\times B}$ and shift $M_{A\times B}$ to the right by 1 or 2 bits until it is normalized.

Rounding Modes

The IEEE specification defines four rounding modes that an implementation must support. These rounding modes are

- Round toward 0.
- Round toward $-\infty$.
- Round toward ∞.
- Round toward nearest.

The specification defines these modes with specific references to bitwise rounding methods that we will not discuss here, but the basic ideas are quite simple. We break the mantissa into the part that can be represented (the leading 1 along with the next 23 most significant bits), which we call M, and the remaining lower-order bits, which we call R. Round toward 0 is also known as *chopping* and is the simplest to understand; in this mode, M is used and R is simply ignored or "chopped off." Round toward $\pm\infty$ are modes that round toward positive (∞) or negative ($-\infty$) based on the sign of the result and whether $R = 0$ or not, as shown in the following tables.

Round toward ∞		
	$R = 0$	$R \neq 0$
$M \geq 0$	M	$M + 1$
$M < 0$	M	M

Round toward $-\infty$		
	$R = 0$	$R \neq 0$
$M \geq 0$	M	M
$M < 0$	M	$M + 1$

1.5.4 Special Values

One of the most important parts of the IEEE floating-point specification is its definition of numerous special values. While these special values co-opt bit patterns that would otherwise represent specific floating-point numbers, this trade-off is accepted as worthwhile, owing to the nature and importance of these special values.

Zero

The representation of 0.0 in floating-point is more complex than one might think. Since the high-order bit of the mantissa is assumed to be 1 (and has no explicit bit in the representation), it is not enough to simply set the 23 explicit mantissa bits to zero, as that would simply represent the number $1.0 \times 2^{exponent-127}$. It is necessary to define zero explicitly, in this case as a number whose exponent bits are all 0 *and* whose explicit mantissa bits are 0. This is sensible, as this value would otherwise represent the smallest possible normalized value. Note that the exponent bits of 0 map to an exponent value of -127, which is reserved for special values such as zero. All other numbers with exponent value -127 (i.e., those with nonzero mantissa bits) are reserved for a class of very small numbers called *denormals*, which will be described later.

Another issue with respect to floating-point zero arises from the fact that IEEE floating-point numbers have an explicit sign bit. The IEEE specification defines both positive and negative 0, differentiated by only the sign bit. To avoid very messy code, the specification does require that floating-point comparisons of positive zero to negative zero return "equal." However, the bitwise representations are distinct, which means that applications should never use bitwise equality tests with floating-point numbers! The bitwise representations of both zeros are

$+0.0 =$

0	00000000	00000000000000000000000
S	Exponent	Mantissa

$-0.0 =$

1	00000000	00000000000000000000000
S	Exponent	Mantissa

The standard does list the behavior of positive and negative zero explicitly, including the definitions:

$$(+0) - (+0) = (+0)$$
$$-(+0) = (-0)$$

Also, the standard defines the sign of the result of a multiplication or division operation as negative if and only if exactly one of the signs of the operands is negative. This includes zeros. Thus,

$$(+0)(+0) = +0$$
$$(-0)(-0) = +0$$

$$(-0)(+0) = -0$$
$$(-0)P = -0$$
$$(+0)P = +0$$
$$(-0)N = +0$$
$$(+0)N = -0$$

where $P > 0$ and $N < 0$.

Infinity

At the other end of the spectrum from zero, the standard also defines positive infinity (∞_{fp}) and negative infinity ($-\infty_{fp}$), along with rules for the behavior of these values. In a sense the infinities are not pure mathematical values. Rather, they are used to represent values that fall outside of the range of valid

exponents. For example, 1.0×10^{38} is just within the range of single-precision floating-point, but in single precision,

$$(1.0 \times 10^{38})^2 = 1.0 \times 10^{76} \approx \infty_{fp}$$

The behavior of infinity is defined by the standard as follows (the standard covers many more cases, but these are representative):

$$\infty_{fp} - P = \infty_{fp}$$
$$\frac{P}{\infty_{fp}} = +0$$
$$\frac{-P}{\infty_{fp}} = -0$$

where

$$0 < P < \infty_{fp}$$

The bitwise representations of $\pm\infty_{fp}$ use the reserved exponent value 128 and all explicit mantissa bits zeros. The only difference between the representations of the two infinities is, of course, the sign bit. The representations are diagrammed as follows:

$\infty_{fp} =$

0	11111111	00000000000000000000000
S	Exponent	Mantissa

$-\infty_{fp} =$

1	11111111	00000000000000000000000
S	Exponent	Mantissa

Floating-point numbers with exponent values of 128 and nonzero mantissa bits do not represent infinities. They represent the next class of special values—*nonnumerics*.

Nonnumeric Values

All the following function call examples represent exceptional cases:

Function Call	**Issue**
arcsine(2.0)	Function not defined for argument.
sqrt(−1.0)	Result is imaginary.
0.0/0.0	Result is indeterminate.
$\infty - \infty$	Result is indeterminate.

In each of these cases, none of the floating-point values we have discussed will accurately represent the situation. Here we need a value that indicates the fact that the desired computation cannot be represented as a real number. The IEEE specification includes a special pair of values for these cases, known collectively as Not a Numbers (NaNs). There are two kinds of NaNs: quiet (or silent) NaN (QNaN) and signaling NaN (SNaN). Compare the following representations:

QNaN =

0	11111111	1[22 low-order bits indeterminate]
S	Exponent	Mantissa

SNaN =

0	11111111	0[22 low-order bits not all 0]
S	Exponent	Mantissa

Quiet Not a Numbers (Kahan [62] simply calls them NaNs) represent indeterminate values and are quietly passed through later computations (generally as QNaNs). They are not supposed to signal an exception, but rather allow floating-point code to return the fact that the result of the desired operation was indeterminate. Floating-point implementations (hardware or software) will generate QNaNs in cases such as those in our comparison.

SNaNs represent unrecoverable mathematical errors and signal an exception. Most floating-point units (FPUs) are designed not to generate SNaNs — the original idea was that authors of high-level software math packages could generate them in terminal situations. In addition, compilers could (in debugging builds) set all floating-point values to SNaN, ensuring an exception if the programmer left the values uninitialized. The realities of compilers and operating systems make SNaNs less interesting. There have been issues in the support for SNaNs in current compilers [62], resulting in SNaNs being encountered very rarely.

1.5.5 Very Small Values

Normalized Mantissas and the "Hole at Zero"

One side effect of the normalized mantissa is very interesting behavior near zero. To better understand this behavior, let us look at the smallest normalized value (we will look at the positive case; the negative case is analogous) in single-precision floating-point, which we will call $F_{min} \cdot F_{min}$. This would have an exponent of -126 and zeros in all explicit mantissa bits. The resulting

mantissa would have only the implicit units bit set, leading to a value of

$$F_{min} = 2^0 \times 2^{-126} = 2^{-126}$$

The largest value smaller than this in a normalized floating-point system would be 0.0. However, the smallest value larger than F_{min} would differ by only 1 bit from F_{min}—the least significant mantissa bit would be set. This value, which we will call F_{next}, would be simply:

$$F_{next} = (2^0 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149} = F_{min} + 2^{-149}$$

This leads to a rather interesting situation: The distance between F_{min} and its nearest smaller neighbor (0.0) is 2^{-126}. This distance is much larger than the distance between F_{min} and its nearest *larger* neighbor, F_{next}. The distance between F_{min} and F_{next} is only

$$F_{next} - F_{min} = 2^{-149}$$

In fact, F_{min} has a sequence of approximately 2^{23} larger neighbors that are each a distance of 2^{-149} from the previous. This leaves a large "hole" of numbers between 0.0 and F_{min} that cannot be represented with nearly the accuracy as the numbers slightly larger than F_{min}. This gap in the representation is often referred to as the *hole at zero*. The operation of representing numbers in the range $(-F_{min}, F_{min})$ with zero is often called *flushing to zero*.

One problem with flush-to-zero is that the subtraction of two numbers that are not equal can result in zero. In other words, with flush-to-zero,

$$A - B = 0 \nRightarrow A = B$$

How can this be? See the following example:

$$A = 2^{-126} \times (2^0 + 2^{-2} + 2^{-3})$$
$$B = 2^{-126} \times (2^0)$$

Both of these are valid single-precision floating-point numbers. In fact, they have equal exponents: −126. Clearly, they are also not equal floating-point numbers: *A*'s mantissa has two additional 1 bits. However, their subtraction produces:

$$\begin{aligned} A - B &= (2^{-126} \times (2^0 + 2^{-2} + 2^{-3}) - (2^{-126} \times (2^0)) \\ &= 2^{-126} \times ((2^0 + 2^{-2} + 2^{-3}) - (2^0)) \\ &= 2^{-126} \times (2^{-2} + 2^{-3}) \\ &= 2^{-128} \times (2^0 + 2^{-1}) \end{aligned}$$

which would be returned as zero on a flush-to-zero floating-point system. While this is a contrived example, it can be seen that any pair of nonequal numbers whose difference has a magnitude less than 2^{-126} would demonstrate this problem. There is a solution to this and other flush-to-zero issues, however. The solution is known as *gradual underflow*, and it is discussed in the next section.

Denormals and Gradual Underflow

The IEEE specification specifies behavior for very small numbers that avoids this so-called hole at zero. The behavior is known as *gradual underflow*, and this gradual underflow generates values called *denormals*, or *denormalized numbers*.

The idea is quite simple. Rather than require every floating-point number to be normalized, the specification reserves numbers with nonzero explicit mantissa bits and an exponent of -127 for denormals. In a denormal, the implicit high-order bit of the mantissa is 0. This allows numbers with magnitude smaller than 1.0×2^{-126} to be represented. In a denormal, the exponent is assumed to be -126 (even though the actual bits would represent -127), and the mantissa is in the range $[\frac{1}{2^{23}}, 1 - \frac{1}{2^{23}}]$. The smallest nonzero value that can be represented with a denormal is $2^{-23} \times 2^{-126} = 2^{-149}$, filling in the "hole at zero." Note that all nonzero floating-point values are still unique, as the specification only allows denormalized mantissas when the exponent is -126, the minimum valid exponent.

As an historical note, gradual underflow and denormalized value handling were perhaps the most hotly contested of all sections in the IEEE floating-point specification. Flush-to-zero is much simpler to implement in hardware, which also tends to mean that it performs faster and makes the hardware cheaper to produce. When the IEEE floating-point standard was being formulated in the late 1970s, several major computer manufacturers were using the flush-to-zero method for dealing with underflow. Changing to the use of gradual underflow required these manufacturers to design FPU hardware or software that could handle the unnormalized mantissas that are generated by denormalization. This would lead either to more complex FPU hardware or a system that emulated some or all of the denormalized computations in software or microcode. The former could make the FPUs more expensive to produce, while the latter could lead to greatly decreased performance of the floating-point system when denormals are generated. However, several manufacturers showed that it could be implemented in floating-point hardware, paving the way for this more accurate method to become part of the de facto (and later, official) standard. However, performance of denormalized values is still an issue, even today. We will discuss a real-world example of denormal performance on a modern FPU in Section 1.6.2.

1.5.6 Catastrophic Cancelation

We have used relative error as a metric of the validity of the floating-point representation of a given number. As we have already seen, converting real numbers A and B into the closest floating-point approximations A_{fp} and B_{fp} generally results in some amount of relative representation error, which we compute as:

$$RelErr_A = \left| \frac{A - A_{fp}}{A} \right|$$

$$RelErr_B = \left| \frac{B - B_{fp}}{B} \right|$$

These relative representation errors accurately represent how well A_{fp} and B_{fp} represent A and B, but the result of adding or subtracting A_{fp} and B_{fp} may contain a much greater level of relative error. The addition or subtraction of a pair of floating-point numbers can lead to a result with magnitude much smaller than either of the operands. Subtracting two nearly (but not exactly) equal values will result in a value whose magnitude is much smaller than either of the operands.

Recall that the last step in adding or subtracting two floating-point numbers is to renormalize the result so that the leading mantissa bit is 1. If the result of an addition or subtraction has much lower magnitude (smaller exponent) than the operands, then there will be some number N of leading mantissa bits that are all zero. The mantissa must be shifted left N bits so that the leading bit is 1 (and the exponent decremented by N, of course), renormalizing the number. Zeros will be shifted into all of the N lowest-order (explicit) mantissa bits. It is these zeros that are the cause of the error, that is, the zeros that are shifted into the lower-order bits are not actual data. Thus, the N least significant mantissa bits may all be wrong. This can greatly compound relative error.

As an example, imagine that we are measuring the distances between pairs of points on the real-number line. Each of these pairs might represent the observed positions of two characters A and B in a game at two different times, t and $t + 1$. We will move each character by the same amount δ between t and $t + 1$. Thus, $A' = A + \delta$ and $B' = B + \delta$. If we use the values

$$A = 1.5$$

$$B = 10^7$$

$$\delta = 1.5$$

We can clearly see that in terms of real numbers,

$$A' = 3.0$$
$$B' = 10{,}000{,}001.5$$

However, if we look at the single-precision floating-point representations, we get

$$A'_{fp} = 3.0$$
$$B'_{fp} = 10{,}000{,}002.0$$

A' is represented exactly, but B' is not, giving a relative error of representation for B'_{fp} of

$$RelErr_B = \left|\frac{0.5}{10^7}\right| = 5 \times 10^{-8}$$

Quite a small relative error. However, if we compute the distances $A' - A$ and $B' - B$ in floating-point, the story is very different:

$$A'_{fp} - A_{fp} = 3.0 - 1.5 = 1.5 = \delta$$
$$B'_{fp} - B_{fp} = 10{,}000{,}002.0 - 10^7 = 2.0$$

In the case of $A' - A$, we get the expected value, δ. But in the case of $B' - B$, we get a relative error of

$$RelErr = \left|\frac{2.0 - 1.5}{1.5}\right| = 0.\overline{3}$$

The resulting error is much larger in the B case, even though $A' - A = B' - B$. What is happening here can be seen by looking at the bitwise representations:

	Exponent	Mantissa Bits
B =	23	10011000100101101000000
B' =	23	10011000100101101000001**0**
$B' - B$ =	23	00000000000000000000010
normalized =	1	10000000000000000000000

In the case of $B' - B$, almost all of the original mantissa bits in the operands were canceled out in the subtraction, leaving the least significant bits of the operands as the *most* significant bit of the result. Basically none of the fractional bits of the resulting mantissa were actual data — the system simply shifted in zeros. The precision of such a result is very low, indeed. This is catastrophic cancelation; the significant bits are all canceled, causing a catastrophically large growth in the representation error of the result.

The best way to handle catastrophic cancelation in a floating-point system is to avoid it. Numerical methods that involve computing a small value as the subtraction or addition of two potentially large values should be reformulated to remove the operation. An example of a common numerical method that uses such a subtraction is the well-known quadratic formula:

$$\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

Both of the subtractions in the numerator can involve large numbers whose addition/subtraction can lead to small results. However, refactoring of the formula can lead to better-conditioned results. The following revised version of the quadratic formula can be used in cases where computation of one of the two roots involves subtracting nearly equal values. The refactored formula avoids cancelation by replacing the subtraction with an addition:

$$\frac{2C}{-B \mp \sqrt{B^2 - 4AC}}$$

A root that would be computed with a subtraction in the first ("classic") version of the quadratic formula may be computed with an addition in the second version, and vice versa.

1.5.7 Double Precision

As mentioned, the IEEE 754 specification supports a 64-bit "double-precision" floating-point value, known in C/C++ as the intrinsic `double` type. The format is completely analogous to the single-precision format, with the following bitwise layout:

Sign	Exponent	Mantissa
1 bit	11 bits	52 bits

Double-precision values have a range of approximately 10^{308} and can represent values smaller than 10^{-308}. A programmer's common response to the

onset of precision or range issues is to switch the code to use double-precision floating-point values in the offending section of code (or sometimes even throughout the entire system). While double precision can solve almost all range issues and many precision issues (though catastrophic cancelation can still persist) in interactive 3D applications, there are several drawbacks that should be considered prior to its use:

- *Memory*. Since double-precision values require twice the storage of single precision values, memory requirements for an application can grow quickly, especially if arrays of values must be stored as double precision.
- *Performance*. At least some of the operations on most hardware FPUs are significantly slower when computing double-precision results. Additional expense can be incurred for conversion between single- and double-precision values.
- *Platform issues*. Not all platforms (especially game-centric platforms) support double precision.

1.6 Real-World Floating-Point

While the IEEE floating-point specification does set the exact behavior for a wide range of the possible cases that occur in real-world situations, in real-world applications on real-world platforms, the specification cannot tell the entire story. The following sections will discuss some issues that are of particular interest to 3D game developers.

1.6.1 Internal FPU Precision

Some readers will likely try some of the exceptional cases themselves in small test applications. In doing so, they are likely to find surprising behavior in many situations. For example, examine the following code:

```
main()
{
    float fHuge = 1.0e30f; // valid single precision
    fHuge *= 1.0e38f; // result = infinity
    fHuge /= 1.0e38f; // ????
}
```

Stepping in a debugger, the following will happen on many major compilers and systems:

1. After the initial assignment, `fHuge = 1.0e30`, as expected.
2. After the multiplication, `fHuge =` ∞_{fp}, as expected.
3. After the division, `fHuge = 1.0e30`!

This seems magical. How can the system divide the single value ∞_{fp} and get back the original number? A look at the assembly code gives a hint. The basic steps the compiler generates are as follows:

1. Load `1.0e30` and `1.0e38` into the FPU.
2. Multiply the two loaded values and return ∞_{fp}, keeping the result in the FPU as well.
3. Divide the previous result (still in the FPU) by `1.0e38` (still in the FPU), returning the correct result.

The important item to note is that the result of each computation was both returned and kept in the FPU for later computation. This step is where the apparent "magic" occurs. The FPU (as per the IEEE standard) uses high-precision (sometimes as long as `long double`) registers in the FPU. The conversion to single precision happens during the transfer of values from the FPU into memory. While the returned value in `fBig` was indeed ∞_{fp}, the value retained in the FPU was higher precision and was the correct value, `1.0e68`. When the division occurs, the result is correct, not ∞_{fp}.

However, an application cannot count on this result. If the FPU had to flush the intermediate values out of its registers, then the result of the three lines above would have been quite different. For example, if significant floating-point work had to be computed between the above multiplication and the final division, the FPU might have run out of registers and had to evict the high-precision version of `fHuge`. This can lead to odd behavior differences, sometimes even between optimized and debugging builds of the same source code.

1.6.2 Performance

The IEEE floating-point standard specifies behavior for floating-point systems; it does not specify information regarding performance. Just because a floating-point implementation is correct does not mean that it is fast. Furthermore, the speed of one floating-point operation (e.g., addition) does

not imply much about the speed of another (e.g., square root). Finally, not all input data are to be considered equal in terms of performance. The following sections describe examples of some real-world performance pitfalls found in floating-point implementations.

Performance of Denormalized Numbers

During the course of creating a demo for a major commercial 3D game engine, one of the authors found that in some conditions, the performance of the demo dropped almost instantaneously by as much as 20 percent. The code was profiled and it was found that one section of animation code was suddenly running 10 to 100 times slower than in the previous frames. An examination of the offending code determined that it consisted of nothing more than basic floating-point operations, specifically, multiplications and divisions. Moreover, there were no loops in the code, and the number of calls to the code was not increasing. The code itself was simply taking 10 to 100 times longer to execute.

Further experiments outside of the demo found that a fixed set of input data (captured from tests of the demo) could always reproduce the problem. The developers examined the code more closely and found that very small nonzero values were creeping into the system. In fact, these numbers were denormalized. Adjusting the numbers by hand even slightly outside of the range of denormals and into normalized floating-point values instantly returned the performance to the original levels. The immediate thought was that exceptions were causing the problem. However, all floating-point exceptions were disabled (masked) in the test application.

To verify the situation, the developers wrote an extremely simple test application. Summarized, it was as follows:

```
float TestFunction(float fValue)
{
    return fValue;
}

main()
{
    int i;
    float fTest;
    // Start "normal" timer here
    for (i = 0; i < 10000; i++)
    {
        // 1.0e-36f is normalized in single precision
        fTest = TestFunction(1.0e-36f);
    }
```

```
        // End "normal" timer here
        // Start "denormal" timer here
        for (i = 0; i < 10000; i++)
        {
            // 1.0e-40f is denormalized in single precision
            fTest = TestFunction(1.0e-40f);
        }
        // End "denormal" timer here
    }
```

Having verified that the assembly code generated by the optimizer did indeed call the desired function the correct number of times with the desired arguments, they found that the denormal loop took 30 times as long as the normal loop (even with exceptions masked). A careful reading of Intel's performance recommendations [60] for the Pentium series of central processing units (CPUs) found that *any* operation (including simply loading to a floating-point register) that produced or accepted as an operand a denormal value was run using so-called assist microcode, which is known to be much slower than standard FPU instructions. Intel's recommendation was for high-performance code to manually clamp small values to zero as need be.

Intel had followed the IEEE 754 specification, but had made the design decision to allow exceptional cases such as denormals to cause very significant performance degradation. An application that had not known of this slowdown on the Pentium processor may have avoided manually clamping small values to zero, out of fear of slowing the application down with extra conditionals. However, armed with this processor-specific information, it was much easier to justify clamping small numbers that were not already known to be normal. The values in question were known to be normalized to be between 0.0 and 1.0. As a result, it was more than safe to clamp small values to zero.

Software Floating-Point Emulation

Applications should take extreme care on new platforms to determine whether or not the platform supports hardware-assisted floating-point. In order to ensure that code from other platforms ports and executes without major rewriting, some compilers supply software floating-point emulation libraries for platforms that do not support floating-point in hardware. This is especially common on popular embedded and handheld chip sets such as Intel's StrongARM and XScale processors [60]. These processors have no FPUs, but C/C++ floating-point code compiled for these devices will generate valid, working emulation code. The compilers will often do this silently, leaving the

uninformed developer with a working program that exhibits horrible floating-point performance, in some cases hundreds of times slower than could be expected from a hardware FPU.

It's worth reiterating that not all FPUs support both single and double precision. Some major game consoles, for example, will generate FPU code for single-precision values and emulation code for double-precision values. As a result, careless use of double precision can lead to much slower code. In fact, it is important to remember that double precision can be introduced into an expression in subtle ways. For example, remember that in C/C++, floating-point constants are double precision by default, so whenever possible, explicitly specify constants as single precision, using the `f` suffix. The difference between double- and single-precision performance can be as simple as `1.0` instead of `1.0f`.

1.6.3 IEEE Specification Compliance

While major floating-point errors in modern processors are relatively rare (even Intel was caught off guard by the magnitude of public reaction to what it considered minor and rare errors in the floating-point divider on the original Pentium chips), this does not mean that it is safe to assume that all floating-point units in modern CPUs are always fully compliant to IEEE specifications and support both single and double precision. The greatest lurking risk to modern developers assuming full IEEE compliance is conscious design decisions, not errors on the part of hardware engineers. However, in most cases, for the careful and attentive programmer, these new processors offer the possibilities of great performance increases to 3D games.

As more and more FPUs are designed and built for multimedia and 3D applications (rather than the historically important scientific computation applications for which earlier FPUs were designed), manufacturers are starting to deviate from the IEEE specification, optimizing for high performance over accuracy. This is especially true with respect to the "exceptional" cases in the spec, such as denormals, infinity, and Not a Numbers.

Hardware vendors make the argument that while these special values are critically important to scientific applications, for 3D games and multimedia they generally occur only in error cases that are best handled by avoiding them in the first place.

Intel's SSE

An important example of such design decisions involves Intel's Streaming SIMD Extensions (SSE) [60], a new coprocessor that was added to the Pentium series with the advent of the Pentium III. The coprocessor is a

special vector processor that can execute parallel math operations on four floating-point values, packed into a 128-bit register. The SSE instructions were specifically targeted at 3D games and multimedia, and this is evident from even a cursory view of the design. Several design decisions related to the special-purpose FPU merit mentioning here:

- The original SSE (Pentium III) instructions can only support 32-bit floating-point values, not doubles.
- Denormal values can be (optionally) rounded to zero (flushed to zero), disabling gradual underflow.
- Full IEEE 754 behavior can be supported as an option but at less than peak performance.

3D-Specific FPUs

Other platforms have created graphics-centric FPUs. This 3D graphics focus has given hardware designers the ability to optimize the floating-point behavior of the FPUs very heavily. Unburdened by the need to support any applications other than games, the designers of these FPUs have taken things a step further than Intel's SSE instructions by making the deviations from the IEEE specification permanent, rather than optional.

AMD's 3DNow! [1] extensions to its x86 platforms are one such example. While leaving the main FPU unchanged, AMD added hardware to support up to four floating-point instructions per clock cycle. As a further optimization, 3DNow! made some decisions that allow implementers (AMD and others) to break from the IEEE specification, including:

- Cannot accept infinity or NaN as operands.
- Generates the maximal normal floating-point value on overflow, rather than infinity.
- Flush-to-zero as the only form of underflow (no denormals).
- Single rounding mode (differs by implementation).
- No support for floating-point exceptions.

The 3D-centric vector FPUs in some current game consoles have taken similar paths. These differences from the IEEE specification, while severe from a scientific computing perspective, are rarely an issue in correct 3D game code. The console processors that have these limitations are generally designed to allow games to implement geometry pipelines. In most 3D game code, the engine programmer takes great pains to avoid exceptional

conditions in the geometry pipelines. Thus, these hardware design decisions tend to merely reflect the common practices of game programmers, rather than adding new limitations upon them.

1.6.4 Graphics Processing Units and Half-Precision Floating-Point Formats

"Half-precision" or `fp16` floating-point numbers refer to a de facto standard format for floating-point values that can fit in 16 bits of storage. While not specified by the IEEE 754 specification and not supported by major CPUs, `fp16` is a format with rapidly growing importance owing to its near ubiquity in modern graphics processing units (GPUs). Most current major GPUs support 16-bit floating-point values in their geometry and pixel rendering pipelines. Since GPUs often handle large amounts of parallel computations on large datasets, minimizing memory usage and bus traffic is an important performance consideration. Being half the size of single precision IEEE 754 floating-point numbers is a significant optimization.

Although `fp16` is not currently a part of the IEEE floating-point standard, the OpenGL and OpenGL ES API standards (or extensions to these standards) [64] both define a format for them that has become basically universal. This format consists of a sign bit, 5 bits of exponent, and 10 bits of mantissa. It should be noted that some GPU vendors also briefly supported a 24-bit floating-point format (sign bit, 7 bits of exponent, and 16 bits of mantissa), but this format is somewhat less standard than the IEEE or `fp16` formats.

The exponent in an `fp16` value varies between either $[-14, 16]$ or $[-14, 15]$, depending on the implementation. The main difference between the two implementations is the maximum exponent value; according to the OpenGL (ES) specification, the format may either use the maximal exponent value of 31 (16 once unbiased) normally or it may reserve it for the `fp16` equivalent of `NaNs` and `INF`. If IEEE-style specials are not supported by an implementation, overflow cases that would have mapped to the special value `INF` are generally clamped to the maximal exponent and mantissa. This ambiguity in the specification was in actuality a reflection of the state of GPUs at the time the specification was created; GPUs implementing both behaviors existed.

In addition, the OpenGL specification of `fp16` values allows for either denormals or flush-to-zero behavior when the exponent is 0 (-15 once unbiased).

While support for both IEEE-style specials and denormals is becoming very common on current GPU `fp16` support, it is not universal, especially if your application must run on older GPUs as well. Applications need to be mindful that very small and very large values may be handled differently on different GPU platforms, not unlike the pre-IEEE 754 floating-point situation. A discussion of how a real application had to deal with these differences in

exceptional behaviors may be found in "GPU Image Processing in Apple's Motion" in Pharr [92].

The reduced size of the `fp16` comes with significantly reduced precision and range when compared to even a single-precision 32-bit floating-point format. Assuming IEEE-style specials and denormals, the extrema of `fp16` are:

$$\begin{aligned} \text{Maximum representable value:} &\quad 65{,}504 \\ \text{Smallest positive value:} &\quad 2^{-25} \approx 3.0 \times 10^{-8} \\ \text{Largest consecutive integer:} &\quad 2{,}048 \end{aligned}$$

These limits can be reached with surprising ease if one is not careful, especially when considering the squaring of numbers. The square of any value greater than around 255 will map to infinity when stored as an `fp16`. And if one is using an `fp16` as a form of integer storage, repeated incrementation will fail at a value that is well within common usage: 2,048. Above 2,048, odd integers simply cannot be represented, with these holes in the set of integers getting larger and larger as the value rises. Thus, `fp16` values are not recommended for counting items. Some of the issues associated with these reduced limits may be found in the article "A Toolkit for Computation on GPUs" in Fernando [36].

How then are `fp16` values usable? The answer is one of real-world use cases. `fp16` values are most frequently used on GPUs in shader code that generates the final color drawn to the screen. In these cases, color-related values have historically been limited to values between 0.0 and 1.0. The ability to use the much larger range afforded by `fp16` values is a relative luxury, and makes such high-end rendering effects as bright light "blooming," glare, and other so-called high dynamic range (HDR) effects possible. The `fp16` format is likely to continue to be popular for some time in GPU code, even though newer GPUs also support single-precision IEEE 754 32-bit floating-point values as well.

1.7 Code

While this text's companion CD-ROM and website do not include specific code that demonstrates the concepts in this chapter, source code that deals with issues of floating-point representation may be found throughout the math library `IvMath`. For example, the source code for `IvMatrix33`, `IvMatrix44`, `IvVector3`, `IvVector4`, and `IvQuat` includes sections of code that avoid denormalized numbers and comparisons to exact floating-point zero.

CPU chipset manufacturers Intel and AMD have been focused on 3D graphics and game performance and have made public many code examples, presentations, and software libraries that detail how to write high-performance floating-point code for their processors. Many of these resources may be found on their developer websites ([1, 60]).

1.8 Chapter Summary

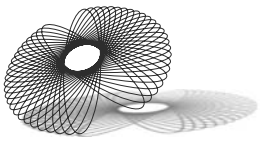
In this chapter we have discussed the details of how computers represent real numbers. These representations have inherent limitations that any serious programmer must understand in order to use them efficiently and correctly. Floating-point presents subtle limitations, especially issues of limited precision. We have also discussed the basics of error metrics for number representations.

Hopefully, this chapter has instilled two important pieces of information in the reader. The first and most basic is an understanding of the inner workings of the number systems that pervade 3D games. This should allow the programmer to truly comprehend the reasons *why* the math-related code behaves (or, more importantly, why it *misbehaves*) as it does. The second piece of information is an appreciation of why one should pay attention to the topic of floating-point representation in the first place — namely, to better prepare the 3D game developer to do what is needed to do at some point in the development of a game: optimize or fix a section of slow or incorrect math code. Better yet, it can assist the developer to avoid writing this potentially problematic code in the first place.

For further reading, Kahan's papers on the history and status of the IEEE floating-point standard ([62] and related papers and lectures by Kahan, available from the same source) offer fascinating insights into the background of modern floating-point computation. In addition, back issues of *Game Developer* magazine (such as [57]) provide frequent discussion of number representations as they relate to computer games.

This page intentionally left blank

Chapter 2



Vectors and Points

2.1 Introduction

The two building blocks of most objects in our interactive digital world are points and vectors. Points represent locations in space, which can be used either as measurements on the surface of an object to approximate the object's shape (this approximation is called a *model*), or as simply the position of a particular object. We can manipulate an object indirectly through its position or by modifying its points directly. Vectors, on the other hand, represent the difference or displacement between two points. Both have some very simple properties that make them extremely useful throughout computer graphics and simulation.

In this chapter we'll discuss the properties and representation of vectors and points, as well as the relationship between them. We'll present how they can be used to build up other familiar entities from geometry classes: in particular, lines, planes, and polygons. Because many problems in computer games boil down to examples in applied algebra, having computer representations of standard geometric objects built on basic primitives is extremely useful.

It is likely that the reader has a basic understanding of these entities from basic math classes but the symbolic representations used by the mathematician may be unfamiliar or forgotten. We will review them in detail here. We will also cover linear algebra concepts — properties of vectors in particular — that are essential for manipulating three-dimensional (3D) objects. Without a thorough understanding of this fundamental material, any work in programming 3D games and applications will be quite confusing.

2.2 Vectors

One might expect that we would cover points first since they are the building blocks of our standard model, but in actuality the basic unit of most of the mathematics we'll discuss in this book is the vector. We'll begin by discussing the vector as a geometric entity since that's primarily how we'll be using it, and it's more intuitive to think of it that way. From there we'll present how we can represent vectors algebraically and how that allows us to manipulate them in the computer. We'll then move to discussing operations that we can perform on vectors and how we can use them to solve certain problems in 3D programming. We'll conclude by formalizing our discussion of vectors by presenting a set of vectors known as a vector space, as well as its algebraic properties.

2.2.1 Geometric Vectors

A *geometric vector* $\mathbf{v}$ is an entity with magnitude (also called length) and direction and is represented graphically as a line segment with an arrowhead on one end (Figure 2.1). The length of the segment represents the magnitude of the vector, and the arrowhead indicates its direction. A vector whose magnitude is 1 is a *unit* or *normalized* vector and is shown as $\hat{\mathbf{v}}$. The *zero* vector $\mathbf{0}$ has a magnitude of zero but no direction.

Note that a vector does not have a location. To make some geometric calculations easier to understand, we may draw two vectors as if they were attached or place a vector relative to a location in space. Despite this, it is important to remember that two vectors with the same magnitude and direction are equal, no matter where they are drawn on a page. For example, in Figure 2.1 the left-most and right-most vectors are equal.

In games we use vectors in one of two ways. The first is as a representation of direction. For example, a vector may indicate direction toward an enemy, toward a light, or perpendicular to a plane. The second meaning represents

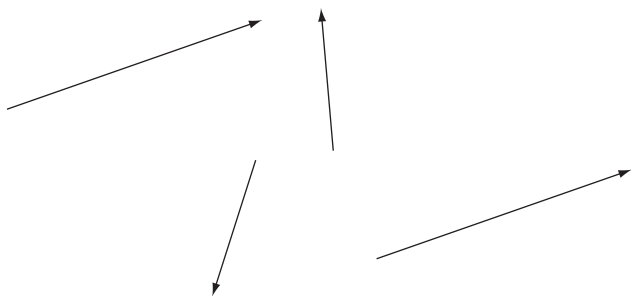


Figure 2.1 Vectors.

change. If we have an object moving through space, we can assign a velocity vector to the object, which represents a change in position. We can displace the object by adding the velocity vector to the object's location to get a new location. Vectors also can be used to represent change in other vectors. For example, we can modify our velocity vector by adding another to it; the second vector is called *acceleration*.

We can perform arithmetic operations on vectors just as we can with real numbers. One basic operation is addition. Geometrically, addition combines two vectors together into a new vector. If we think of a vector as an agent that changes position, then the new vector $\mathbf{u} = \mathbf{v} + \mathbf{w}$ combines the position-changing effect of $\mathbf{v}$ and $\mathbf{w}$ into one entity.

As an example, in Figure 2.2 we have three locations P, Q, and R. There is a vector $\mathbf{v}$ that represents the change in position or displacement from P to Q and a vector $\mathbf{w}$ that represents the displacement from Q to R. If we want to know the vector that represents the displacement from P to R, then we add $\mathbf{v}$ and $\mathbf{w}$ to get the resulting vector $\mathbf{u}$.

Figure 2.3 shows another approach, which is to treat the two vectors as the sides of a parallelogram. In this case, the sum of the two vectors is the long diagonal of the parallelogram. Subtraction, or $\mathbf{v} - \mathbf{w}$, is shown by the other vector crossing the parallelogram. Remember that the difference vector is drawn from the second vector head to the first vector head — the opposite of what one might expect.

The algebraic rules for vector addition are very similar to real numbers:

1. $\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$ (commutative property).
2. $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ (associative property).
3. $\mathbf{v} + \mathbf{0} = \mathbf{v}$ (additive identity).
4. For every $\mathbf{v}$, there is a vector $-\mathbf{v}$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$ (additive inverse).

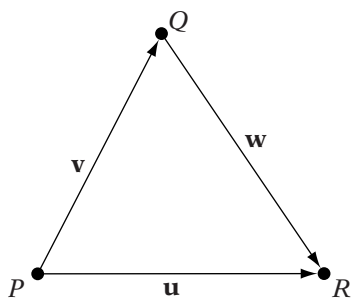


FIGURE 2.2 Vector addition.

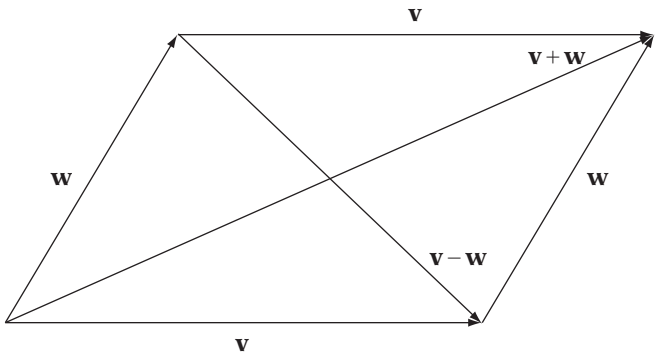


FIGURE 2.3 Vector addition and subtraction.

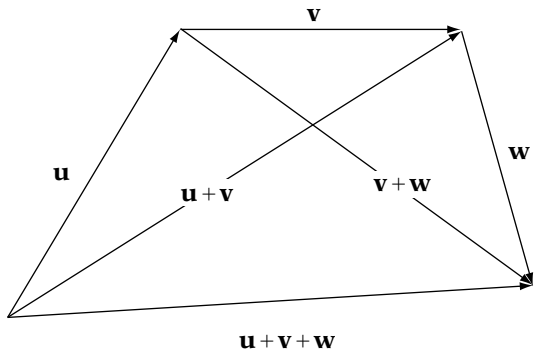


FIGURE 2.4 Associative property of vector addition.

We can verify this informally by drawing a few test cases. For example, if we examine Figure 2.3 again, we can see that one path along the parallelogram represents $\mathbf{v} + \mathbf{w}$ and the other represents $\mathbf{w} + \mathbf{v}$. The resulting vector is the same in both cases. Figure 2.4 presents the associative property in a similar fashion.

The other basic operation is scalar multiplication, which changes the length of a vector by multiplying it by a single real value, also known as a *scalar* (Figure 2.5). Multiplying a vector by 2, for example, makes it twice as long. Multiplying by a negative value changes the length and points the vector in the opposite direction (the length remains nonnegative). Multiplying by 0 always produces the zero vector $\mathbf{0}$.

The algebraic rules for scalar multiplication should also look familiar:

5. $(ab)\mathbf{v} = a(b\mathbf{v})$ (associative property).
6. $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$ (distributive property).

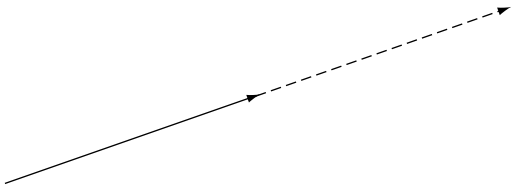


FIGURE 2.5 Scalar multiplication.

7. $a(\mathbf{v} + \mathbf{w}) = a\mathbf{v} + a\mathbf{w}$ (distributive property).
8. $1 \cdot \mathbf{v} = \mathbf{v}$ (multiplicative identity).

As with the additive rules, diagrams can be created that provide a certain amount of intuitive understanding.

2.2.2 Linear Combinations

Our definitions of vector addition and scalar multiplication can be used to describe some special properties of vectors. Suppose we have a set S of n vectors, where $S = \{\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}\}$. We can combine these to create a new vector $\mathbf{v}$ using the function

$$\mathbf{v} = a_0\mathbf{v}_0 + a_1\mathbf{v}_1 + \cdots + a_{n-1}\mathbf{v}_{n-1}$$

for some arbitrary real scalars $a_0, \ldots, a_{n-1}$. This is known as a *linear combination* of all vectors $\mathbf{v}_i$ in S.

If we take all the possible linear combinations of all vectors in S, then the set T of vectors thus created is the *span* of S. We can also say that the set S spans the set T. For example, vectors $\mathbf{v}_0$ and $\mathbf{v}_1$ in Figure 2.6 span the set of vectors that lie on the surface of the page (assuming your book is held flat).

We can use linear combinations to define some properties of our initial set S. Suppose we can find a single nonzero vector $\mathbf{v}_i$ in S such that it's equal to a linear combination of other members of S. In other words,

$$\mathbf{v}_i = a_0\mathbf{v}_0 + \cdots + a_{i-1}\mathbf{v}_{i-1} + a_{i+1}\mathbf{v}_{i+1} + \cdots + a_{n-1}\mathbf{v}_{n-1}$$

If such a $\mathbf{v}_i$ exists, then we say that S is *linearly dependent*. If we can't find any such $\mathbf{v}_i$, then the vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}$ are *linearly independent*. An example of a linearly dependent set of vectors can be seen in Figure 2.7. Vector $\mathbf{v}_0$ is equal to the linear combination $-1 \cdot \mathbf{v}_1 + 0 \cdot \mathbf{v}_2$, or just $-\mathbf{v}_1$. Two linearly dependent vectors $\mathbf{v}$ and $\mathbf{w}$ are said to be *parallel*; that is, $\mathbf{w} = a\mathbf{v}$.

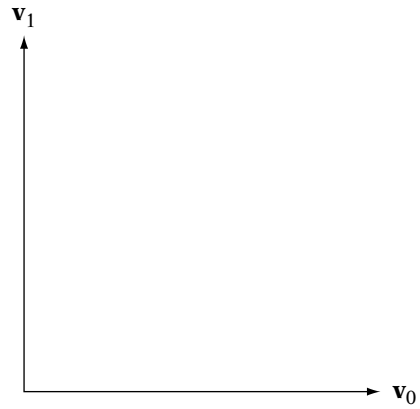


FIGURE 2.6 Two vectors spanning a plane.

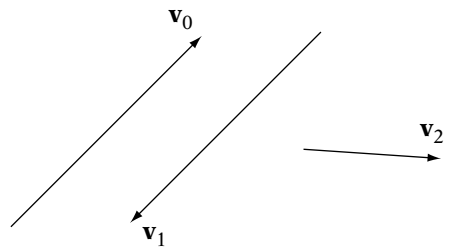


FIGURE 2.7 Linearly dependent set of vectors.

2.2.3 VECTOR REPRESENTATION

In symbolic mathematics and (more important for our purposes) in the computer, representing vectors graphically is not convenient. Instead we define a set of linearly independent vectors known as a *basis*, and define our remaining vectors in terms of those. So for example, for 3D space (formally represented as $\mathbb{R}^3$) we can define three vectors **i**, **j**, and **k**. Their corresponding geometric representations can be seen in Figure 2.8. Note that these vectors are of unit length and perpendicular to each other (we will define "perpendicular" more formally when we discuss dot products). This is known as the *standard Euclidean basis*. Using this basis, we can uniquely represent any vector **v** in 3D space by using the linear combination

$$\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

We can think of x, y, and z as the amounts we move in the **i**, **j**, and **k** directions, from the tail of **v** to its tip (see Figure 2.8). Since the **i**, **j**, and **k** vectors are

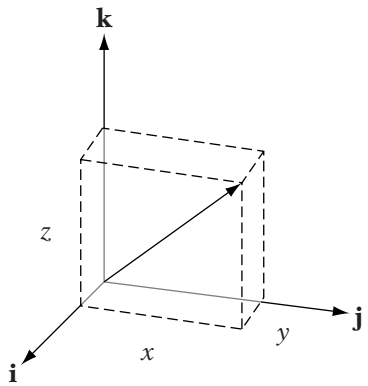


FIGURE 2.8 Standard 3D basis vectors.

known and fixed, we just store the x, y, and z values and use them to represent our vector numerically. In this way a 3D vector $\mathbf{v}$ is represented by an ordered triple (x, y, z). These are known as the vector *components*. Our basis vectors $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ will be represented with components (1, 0, 0), (0, 1, 0), and (0, 0, 1), respectively.

We can do the same for two-dimensional (2D) space, or $\mathbb{R}^2$, by using as our basis $\{\mathbf{i}, \mathbf{j}\}$, where $\mathbf{i} = (1, 0)$ and $\mathbf{j} = (0, 1)$, and representing a 2D vector as the ordered pair (x, y).

By doing this, we have also neatly solved the problem of representing our geometric vectors algebraically. By using a standard basis, we can use an ordered triple to represent the same concept as a line segment with an arrowhead. And by setting a correspondence between our algebraic basis and our geometric representation, we can guarantee that the ordered triple we use in one circumstance will be the same as the one we use in the other. Because of this, when working with vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$, we will use the two representations interchangeably.

Using our new representation allows us to define addition and scalar multiplication algebraically. So, if we add two 3D vectors $\mathbf{v}_0$ and $\mathbf{v}_1$ together and expand and rearrange terms, we get

$$\begin{aligned}\mathbf{v}_0 + \mathbf{v}_1 &= (x_0\mathbf{i} + y_0\mathbf{j} + z_0\mathbf{k}) + (x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k}) \\ &= x_0\mathbf{i} + x_1\mathbf{i} + y_0\mathbf{j} + y_1\mathbf{j} + z_0\mathbf{k} + z_1\mathbf{k} \\ &= (x_0 + x_1)\mathbf{i} + (y_0 + y_1)\mathbf{j} + (z_0 + z_1)\mathbf{k}\end{aligned}$$

If we remove $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ to create ordered triples, we find that to add two vectors we take each component in xyz order and add them:

$$(x_0, y_0, z_0) + (x_1, y_1, z_1) = (x_0 + x_1, y_0 + y_1, z_0 + z_1) \tag{2.1}$$

Scalar multiplication works similarly:

$$\begin{aligned} a\mathbf{v} &= a(x\mathbf{i} + y\mathbf{j} + z\mathbf{k}) \\ &= a(x\mathbf{i}) + a(y\mathbf{j}) + a(z\mathbf{k}) \\ &= (ax)\mathbf{i} + (ay)\mathbf{j} + (az)\mathbf{k} \end{aligned}$$

And again, pulling out **i**, **j**, and **k** gives us

$$a(x, y, z) = (ax, ay, az) \tag{2.2}$$

A more formal discussion of basis vectors can be found in Section 2.2.11.

2.2.4 Basic Vector Class Implementation

Now that we've presented an algebraic representation for vectors, we can talk about how we will store them in the computer. In our case we'll assume the standard basis and thus store the components x, y, and z.

The following are some excerpts from the included C++ math library. For a vector in $\mathbb{R}^3$, our bare-bones class definition is

```
class IvVector3
{
   inline IvVector3() {}
   inline IvVector3( float xVal, float yVal, float zVal ) :
      x( xVal ),
      y( yVal ),
      z( zVal )
   {
   }
   inline ~IvVector3() {}
   IvVector3( const IvVector3& vector );
   IvVector3& operator=( const IvVector3& vector );
   inline float GetX() { return x; }
   inline float GetY() { return y; }
   inline float GetZ() { return z; }
   inline void SetX( float xVal ) { x = xVal; }
   inline void SetY( float yVal ) { y = yVal; }
   inline void SetZ( float zVal ) { z = zVal; }
   ...
private:
   float x,y,z;
}
```

We can observe a few things about this declaration. First, we declared our member variables as a type `float`. This is the single-precision IEEE floating-point representation for real numbers, as discussed in Chapter 1. While not as precise as double-precision floating-point, it has the advantage of being compact and compatible with standard representations on most graphics hardware.

The second thing to notice is that, unlike the previous edition of this book, we're making our member variables private and only providing access through inlined member functions. Again, as discussed in Chapter 1, most modern processors have a platform-specific instruction set for vectors — for example, SSE on Pentium and AMD processors, and VMX on the PowerPC, Xeon, and Cell processors — which can perform several floating-point operations in parallel and can lead to significantly faster code. By hiding the underlying data, it is much easier to change and optimize the underlying representation without modifying nonlibrary code. For clarity of code, we have chosen to not use platform-specific operations, but still hide the data as part of good coding practice.

In some cases, it is possible to write the class to allow public access to the components of the platform-specific data. However, even then we recommend using inlined member functions, as accessing the components in this way is often an expensive operation. So, in general it is recommended that whenever possible you modify the platform-specific data only through platform-specific operations. Using member functions helps reinforce this good practice as well.

The class has a default constructor and destructor, which do nothing. The constructor could initialize the components to 0, but doing so takes time, which adds up when we have large arrays of vectors (a common occurrence), and in most cases we'll be setting the values to something else anyway. For this purpose, there is an additional constructor that takes three floating-point values and uses them to set the components. We can use the copy constructor and assignment operator as well.

Now that we have the data set up for our class, we can add some operations to it. The corresponding operator for vector addition is

```
IvVector3 operator+(const IvVector3& v0, const IvVector3& v1)
{
    return IvVector3( v0.x + v1.x, v0.y + v1.y, v0.z  + v1.z );
}
```

Scalar multiplication is also straightforward:

```
IvVector3
operator*( float a, const IvVector3& vector)
```

```
{
    return IvVector3( a*vector.x, a*vector.y, a*vector.z );
}
```

These methods are given friend access by the class to allow use of the private member variables.

Similar operators for postmultiplication and division by a scalar are also provided within the library; their declarations are:

```
IvVector3 operator*( const IvVector3& vector, float scalar );
IvVector3 operator/( const IvVector3& vector, float scalar );
IvVector3& operator*=( IvVector3& vector, float scalar );
IvVector3& operator/=( IvVector3& vector, float scalar );
```

Now that we have a numeric representation for vectors and have covered the algebraic form of addition and scaling, we can add some new vector operations as well. As before, we'll focus primarily on the case of $\mathbb{R}^3$. Vectors in $\mathbb{R}^2$ and $\mathbb{R}^4$ have similar properties; any exceptions will be discussed in the particular parts.

2.2.5 Vector Length

We have mentioned that a vector is an entity with length and direction but so far haven't provided any means of measuring or comparing these quantities in two vectors. We'll see shortly how the dot product provides a way to compare vector directions. First, however, we'll consider how to measure a vector's magnitude.

There is a general class of size-measuring functions known as *norms*. A norm $\|\mathbf{v}\|$ is defined as a real-valued function on a vector $\mathbf{v}$ with the following properties:

1. $\|\mathbf{v}\| \geq 0$, and $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v} = \mathbf{0}$.
2. $\|a\mathbf{v}\| = |a|\|\mathbf{v}\|$.
3. $\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$.

We use the $\|\mathbf{v}\|$ notation to distinguish a norm from the absolute value function $|a|$.

An example of a norm is the Manhattan distance, also called the ℓ_1 norm, which is just the sum of the absolute values of the given vector's components:

$$\|\mathbf{v}\|_{\ell_1} = \sum_i |v_i|$$

One that we'll use more often is the Euclidean norm, also known as the ℓ_2 norm or just *length*. If we give no indication of which type of norm we're using, this is usually what we mean.

We derive the Euclidean norm as follows. Suppose we have a 2D vector $\mathbf{u} = x\mathbf{i} + y\mathbf{j}$. Recall the Pythagorean theorem $x^2 + y^2 = d^2$. Since x is the distance along $\mathbf{i}$ and y is the distance along $\mathbf{j}$, then the length d of $\mathbf{u}$ is

$$\|\mathbf{u}\| = d = \sqrt{x^2 + y^2}$$

as shown in Figure 2.9. A similar formula is used for a vector $\mathbf{v} = (x, y, z)$, using the standard basis in $\mathbb{R}^3$:

$$\|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2} \tag{2.3}$$

And the general form in $\mathbb{R}^n$ with respect to the standard basis is

$$\|\mathbf{v}\| = \sqrt{v_0^2 + v_1^2 + \cdots + v_{n-1}^2}$$

We've mentioned the use of unit length vectors as pure indicators of direction, for example, in determining viewing direction or relative location of a light source. Often, though, the process we'll use to generate our direction vector will not automatically create one of unit length. To create a unit vector $\hat{\mathbf{v}}$ from a general vector $\mathbf{v}$, we *normalize* $\mathbf{v}$ by multiplying it by 1 over its length, or

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

This sets the length of the vector to $\|\mathbf{v}\| \cdot 1/\|\mathbf{v}\|$ or, as we desire, 1.

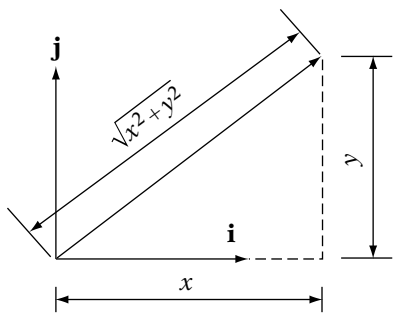


FIGURE 2.9 Length of 2D vector.

Our implementations of length methods (for $\mathbb{R}^3$) are as follows:

```
float
IvVector3::Length() const
{
    return IvSqrt( x*x + y*y + z*z );
}

float
IvVector3::LengthSquared() const
{
    return x*x + y*y + z*z;
}

IvVector3&
IvVector3::Normalize()
{
    float lengthsq = x*x + y*y + z*z;
    ASSERT( !IsZero( lengthsq ) );
    if ( IsZero( lengthsq ) )
    {
        x = y = z = 0.0f;
        return *this;
    }

    float recip = IvInvSqrt( lengthsq );
    x *= recip;
    y *= recip;
    z *= recip;

    return *this;
}
```

Note that in addition to the mathematical operations we've just described, we have defined a `LengthSquared()` method. Performing the square root can be a costly operation, even on systems that have a special machine instruction to compute it. Often we're only doing a comparison between lengths, so it is better and certainly faster in those cases to compute and compare length squared instead. Both length and length squared are increasing functions starting at 0, so the results will be the same.

The length methods also introduce some new functions that will be useful to us throughout the math library. We use our own square root functions `IvSqrt()` and `IvInvSqrt()` instead of `sqrtf()`. There are a number of reasons for this choice. As mentioned, the standard library implementation of

square root is often slow. Rather than use it, we can use an approximation on some platforms, which is faster and accurate enough for our purpose. On other platforms there are internal assembly instructions that are not used by the standard library. In particular, there may be an instruction that performs the inverse square root, which is faster than calculating the square root and performing the floating-point divide. Defining our own layer of indirection gives us flexibility and ensures that we can guarantee ourselves the best performance.

2.2.6 Dot Product

Now that we've considered vector length, we can look at vector direction. The function we will use for this is called the *dot product*, or less commonly, the *Euclidean inner product* (see below for the formal definition of inner products). It is probably the most useful vector operation for 3D games and applications.

Given two vectors $\mathbf{v}$ and $\mathbf{w}$ with an angle θ between them, the dot product $\mathbf{v} \cdot \mathbf{w}$ is defined as

$$\mathbf{v} \cdot \mathbf{w} = \|\mathbf{v}\| \|\mathbf{w}\| \cos\theta \tag{2.4}$$

Using this equation, we can find a coordinate-dependent definition in $\mathbb{R}^3$ by examining a triangle formed by $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{v} - \mathbf{w}$ (Figure 2.10). The Law of Cosines (see Appendix A on the CD-ROM) gives us

$$\|\mathbf{v} - \mathbf{w}\|^2 = \|\mathbf{v}\|^2 + \|\mathbf{w}\|^2 - 2\|\mathbf{v}\| \|\mathbf{w}\| \cos\theta$$

We can rewrite this as

$$-2\|\mathbf{v}\| \|\mathbf{w}\| \cos\theta = \|\mathbf{v} - \mathbf{w}\|^2 - \|\mathbf{v}\|^2 - \|\mathbf{w}\|^2$$

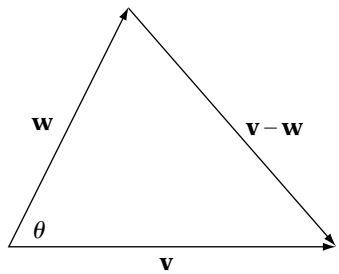


Figure 2.10 Law of Cosines.

Substituting in the definition of vector length in $\mathbb{R}^3$ and expanding, we get

$$\begin{aligned} -2\|\mathbf{v}\|\|\mathbf{w}\|\cos\theta &= (v_x - w_x)^2 + (v_y - w_y)^2 + (v_z - w_z)^2 \\ &\quad - (v_x^2 + v_y^2 + v_z^2) - (w_x^2 + w_y^2 + w_z^2) \\ -2\|\mathbf{v}\|\|\mathbf{w}\|\cos\theta &= -2v_x w_x - 2v_y w_y - 2v_z w_z \\ \|\mathbf{v}\|\|\mathbf{w}\|\cos\theta &= v_x w_x + v_y w_y + v_z w_z \end{aligned}$$

So, to compute the dot product in $\mathbb{R}^3$, multiply the vectors componentwise, and then add:

$$\mathbf{v} \cdot \mathbf{w} = v_x w_x + v_y w_y + v_z w_z$$

Note that for this definition to hold, vectors $\mathbf{v}$ and $\mathbf{w}$ need to be represented with respect to the standard basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$. The general form for vectors $\mathbf{v}$ and $\mathbf{w}$ in $\mathbb{R}^n$, again with respect to the standard basis, is

$$\mathbf{v} \cdot \mathbf{w} = v_0 w_0 + v_1 w_1 + \cdots + v_{n-1} w_{n-1}$$

For vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, and scalar a the following algebraic rules apply:

1. $\mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{v}$ (symmetry).
2. $(\mathbf{u} + \mathbf{v}) \cdot \mathbf{w} = \mathbf{u} \cdot \mathbf{w} + \mathbf{v} \cdot \mathbf{w}$ (additivity).
3. $a(\mathbf{v} \cdot \mathbf{w}) = (a\mathbf{v}) \cdot \mathbf{w} = \mathbf{v} \cdot (a\mathbf{w})$ (homogeneity).[1]
4. $\mathbf{v} \cdot \mathbf{v} \geq 0$ (positivity).
5. $\mathbf{v} \cdot \mathbf{v} = 0$ if and only if $\mathbf{v} = \mathbf{0}$ (definiteness).

Also note that we can relate the dot product to the length function by noting that

$$\mathbf{v} \cdot \mathbf{v} = \|\mathbf{v}\|^2 \tag{2.5}$$

As mentioned, the dot product has many uses. By equation 2.4, if the angle between two vectors $\mathbf{v}$ and $\mathbf{w}$ in standard Euclidean space is 90 degrees, then $\mathbf{v} \cdot \mathbf{w} = 0$. So, we define that two vectors $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, or *orthogonal*, when $\mathbf{v} \cdot \mathbf{w} = 0$. Recall that we stated that our standard basis

1. Note that the leading scalar does not apply to both terms on the right-hand side; assuming so is a common mistake.

vectors for $\mathbb{R}^3$ are orthogonal. We can now demonstrate this. For example, taking $\mathbf{i} \cdot \mathbf{j}$ we get

$$\begin{aligned}\mathbf{i} \cdot \mathbf{j} &= (1, 0, 0) \cdot (0, 1, 0) \\ &= 0 + 0 + 0 \\ &= 0\end{aligned}$$

It is possible, although not always recommended, to use equation 2.4 to test whether two unit vectors $\hat{\mathbf{v}}$ and $\hat{\mathbf{w}}$ are pointing generally in the same direction. If they are, $\cos\theta$ is close to 1, so $1 - \hat{\mathbf{v}} \cdot \hat{\mathbf{w}}$ is close to 0 (we use this formula to avoid problems with floating-point precision). Similarly, if $1 + \hat{\mathbf{v}} \cdot \hat{\mathbf{w}}$ is close to 0, they are pointing in opposite directions. Performing this test only takes 6 floating-point addition and multiplication operations. However, if $\mathbf{v}$ and $\mathbf{w}$ are not known to be normalized, then we need a different test: $\|\mathbf{v}\|^2\|\mathbf{w}\|^2 - (\mathbf{v} \cdot \mathbf{w})^2$. This takes 18 operations.

Note that for unit vectors:

$$\begin{aligned}1 - (\hat{\mathbf{v}} \cdot \hat{\mathbf{w}})^2 &= 1 - \cos^2\theta \\ &= \sin^2\theta\end{aligned}$$

and for nonunit vectors:

$$\begin{aligned}\|\mathbf{v}\|^2\|\mathbf{w}\|^2 - (\mathbf{v} \cdot \mathbf{w})^2 &= \|\mathbf{v}\|^2\|\mathbf{w}\|^2(1 - \cos^2\theta) \\ &= \|\mathbf{v}\|^2\|\mathbf{w}\|^2 \sin^2\theta\end{aligned}$$

So assuming we use this, the method we use to test closeness to zero will have to be different for both cases.

In any case, using dot product for this test is not really recommended unless your vectors are prenormalized *and* speed is of the essence. As $\cos\theta$ gets close to 0, it changes very little. Due to lack of floating-point precision, the set of angles that might be considered "zero" is actually broader than one might expect. As we will see, there is another method to test for parallel vectors that is faster with nonunit vectors and has fewer problems with near-zero angles.

A more common use of the dot product is to test values of the angle between two vectors. We know that if $\mathbf{v} \cdot \mathbf{w} > 0$, then the angle is less than 90 degrees; if $\mathbf{v} \cdot \mathbf{w} < 0$, then the angle is greater than 90 degrees; and if $\mathbf{v} \cdot \mathbf{w} = 0$, then the angle is exactly 90 degrees (Figure 2.11). As opposed to testing for parallel vectors, this will work with vectors of *any* length.

For example, suppose that we have an artificial intelligence (AI) agent that is looking for enemy agents in the game. The AI has a view vector $\mathbf{v}$ and a vector $\mathbf{t}$ that points toward an object in our scene. If $\mathbf{v} \cdot \mathbf{t} < 0$, then the object is behind us and therefore not visible to our AI (Figure 2.12).

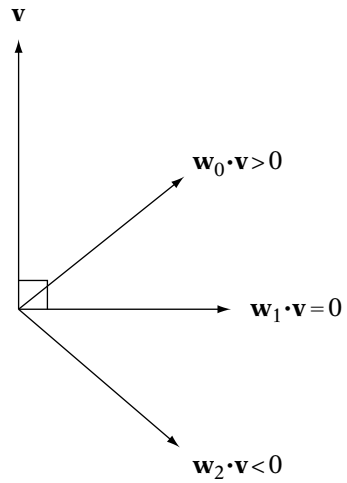


FIGURE 2.11 Dot product as measurement of angle.

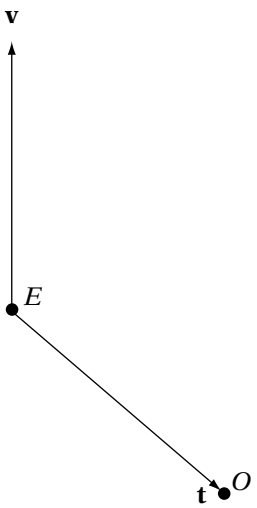


FIGURE 2.12 Measuring angle to target.

Equation 2.4 allows us to use the dot product in another manner. Suppose we have two vectors $\mathbf{v}$ and $\mathbf{w}$, where $\mathbf{w} \neq 0$. We define the *projection* of $\mathbf{v}$ onto $\mathbf{w}$ as

$$\text{proj}_{\mathbf{w}}\,\mathbf{v} = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{w}\|^2}\mathbf{w}$$

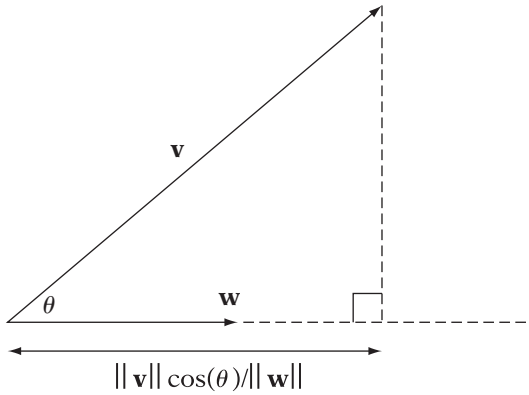


FIGURE 2.13 Dot product as projection.

This gives the part of **v** that is parallel to **w**, which is the same as dropping a perpendicular from the end of **v** onto **w** (Figure 2.13).

We can get the part of **v** that is perpendicular to **w** by subtracting the projection:

$$\text{perp}_{\mathbf{w}}\mathbf{v} = \mathbf{v} - \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{w}\|^2}\mathbf{w}$$

Both of these equations will be very useful to us. Note that if **w** is normalized, then the projection simplifies to

$$\text{proj}_{\hat{\mathbf{w}}}\mathbf{v} = (\mathbf{v}\cdot\hat{\mathbf{w}})\hat{\mathbf{w}}$$

The corresponding library implementation of dot product in $\mathbb{R}^3$ is as follows:

```
float
IvVector3::Dot( const IvVector3& other )
{
   return x*other.x + y*other.y + z*other.z;
}
```

2.2.7 Gram-Schmidt Orthogonalization

The combination of dot product and normalization allows us to define a particularly useful class of vectors. If a set of vectors β are all unit vectors and pairwise orthogonal, we say that they are *orthonormal*. Our standard basis {**i**, **j**, **k**} is an example of an orthonormal set of vectors.

In many cases we start with a general set of vectors and want to generate the closest possible orthonormal one. One example of this is when we perform operations on currently orthonormal vectors. Even if the pure mathematical result should not change their length or relative orientation, due to floating-point precision problems the resulting vectors may be no longer orthonormal. The process that allows us to create orthonormal vectors from possibly nonorthonormal vectors is called *Gram-Schmidt orthogonalization*.

This works as follows. Suppose we have a set of nonorthogonal vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}$, and from them we want to create an orthonormal set $\mathbf{w}_0, \ldots, \mathbf{w}_{n-1}$. We'll use the first vector from our original set as the starting vector for our new set so

$$\mathbf{w}_0 = \mathbf{v}_0$$

Now we want to create a vector orthogonal to $\mathbf{w}_0$, which points generally in the direction of $\mathbf{v}_1$. We can do this by computing the projection of $\mathbf{v}_1$ on $\mathbf{w}_0$, which produces the component vector of $\mathbf{v}_1$ parallel to $\mathbf{w}_0$. The remainder of $\mathbf{v}_1$ will be orthogonal to $\mathbf{w}_0$, so

$$\begin{aligned}\mathbf{w}_1 &= \mathbf{v}_1 - \text{proj}_{\mathbf{w}_0}\mathbf{v}_1 \\ &= \mathbf{v}_1 - \frac{\mathbf{v}_1 \cdot \mathbf{w}_0}{\|\mathbf{w}_0\|^2}\mathbf{w}_0\end{aligned}$$

We perform the same process for $\mathbf{w}_2$: We project $\mathbf{v}_2$ on $\mathbf{w}_0$ and $\mathbf{w}_1$ to compute the parallel components and then subtract those from $\mathbf{v}_2$ to generate a vector orthogonal to both $\mathbf{w}_0$ and $\mathbf{w}_1$:

$$\begin{aligned}\mathbf{w}_2 &= \mathbf{v}_2 - \text{proj}_{\mathbf{w}_0}\mathbf{v}_2 - \text{proj}_{\mathbf{w}_1}\mathbf{v}_2 \\ &= \mathbf{v}_2 - \frac{\mathbf{v}_2 \cdot \mathbf{w}_0}{\|\mathbf{w}_0\|^2}\mathbf{w}_0 - \frac{\mathbf{v}_2 \cdot \mathbf{w}_1}{\|\mathbf{w}_1\|^2}\mathbf{w}_1\end{aligned}$$

In general, we have

$$\begin{aligned}\mathbf{w}_i &= \mathbf{v}_i - \sum_{j=0}^{i-1} \text{proj}_{\mathbf{w}_j}\mathbf{v}_i \\ &= \mathbf{v}_i - \sum_{j=0}^{i-1} \frac{\mathbf{v}_i \cdot \mathbf{w}_j}{\|\mathbf{w}_j\|^2}\mathbf{w}_j\end{aligned}$$

Performing this for all n vectors will give us an orthogonal set of vectors. To create an orthonormal set, we can either normalize the resulting $\mathbf{w}_j$ vectors at the end or normalize as we go, the latter of which simplifies the projection calculation to $(\mathbf{v}_i \cdot \hat{\mathbf{w}}_j)\,\hat{\mathbf{w}}_j$.

2.2.8 Cross Product

Suppose we have two vectors $\mathbf{v}$ and $\mathbf{w}$ and want to find a new vector $\mathbf{u}$ orthogonal to both. The operation that computes this is the *cross product*, also known as the *vector product*. There are two possible choices for the direction of the vector, each the negation of the other (Figure 2.14); the one chosen is determined by the right-hand rule. Hold your right hand so that your forefinger points forward, your middle finger points out to the left, and your thumb points up. If you roughly align your forefinger with $\mathbf{v}$, and your middle finger with $\mathbf{w}$, then the cross product will point in the direction of your thumb (Figure 2.15). The length of the cross product is equal to the area of a parallelogram bordered by the two vectors (Figure 2.16). This can be computed using the formula

$$\|\mathbf{v} \times \mathbf{w}\| = \|\mathbf{v}\|\|\mathbf{w}\| \sin\theta \tag{2.6}$$

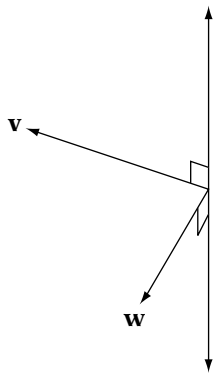


Figure 2.14 Two directions of orthogonal 3D vectors.

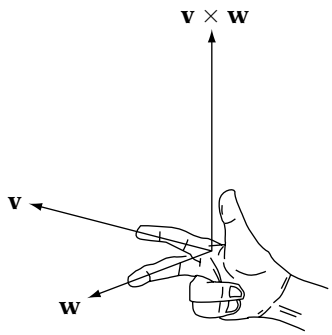


Figure 2.15 Cross product direction.

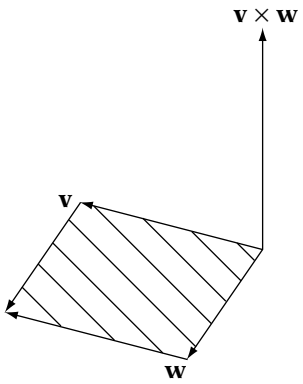


FIGURE 2.16 Cross product length equals area of parallelogram.

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$. Note that the cross product is not commutative, so order is important:

$$\mathbf{v} \times \mathbf{w} = -(\mathbf{w} \times \mathbf{v})$$

Also, if the two vectors are parallel, $\sin\theta = 0$, so we end up with the zero vector.

It is a common mistake to believe that if $\mathbf{v}$ and $\mathbf{w}$ are unit vectors, the cross product will also be a unit vector. A quick look at equation 2.6 shows this is true only if $\sin\theta$ is 1, in which case θ is 90 degrees.

The formula for the cross product is

$$\mathbf{v} \times \mathbf{w} = (v_y w_z - w_y v_z, v_z w_x - w_z v_x, v_x w_y - w_x v_y)$$

Certain processors can implement this as a two-step operation, by creating two vectors and performing the subtraction in parallel:

$$\mathbf{v} \times \mathbf{w} = (v_y w_z, v_z w_x, v_x w_y) - (w_y v_z, w_z v_x, w_x v_y)$$

For vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, and scalar a, the following algebraic rules apply:

1. $\mathbf{v} \times \mathbf{w} = -\mathbf{w} \times \mathbf{v}$.
2. $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) + (\mathbf{u} \times \mathbf{w})$.
3. $(\mathbf{u} + \mathbf{v}) \times \mathbf{w} = (\mathbf{u} \times \mathbf{w}) + (\mathbf{v} \times \mathbf{w})$.
4. $a(\mathbf{v} \times \mathbf{w}) = (a\mathbf{v}) \times \mathbf{w} = \mathbf{v} \times (a\mathbf{w})$.
5. $\mathbf{v} \times \mathbf{0} = \mathbf{0} \times \mathbf{v} = \mathbf{0}$.
6. $\mathbf{v} \times \mathbf{v} = \mathbf{0}$.

There are two common uses for the cross product. The first, and most used, is to generate a vector orthogonal to two others. Suppose we have three points P, Q, and R, and we want to generate a unit vector $\mathbf{n}$ that is orthogonal to the plane formed by the three points (this is known as a normal vector). Begin by computing $\mathbf{v} = (Q - P)$ and $\mathbf{w} = (R - P)$. Now we have a decision to make. Computing $\mathbf{v} \times \mathbf{w}$ and normalizing will generate a normal in one direction, whereas $\mathbf{w} \times \mathbf{v}$ and normalizing will generate one in the opposite direction (Figure 2.17). Usually we'll set things up so that the normal points from the inside toward the outside of our object.

Like the dot product, the cross product can also be used to determine if two vectors are parallel by checking whether the resulting vector is close to the zero vector. Deciding whether to use this test as opposed to the dot product depends on what your data are. The cross product takes 9 operations. We can test for zero by examining the dot product of the result with itself $((\mathbf{v} \times \mathbf{w}) \cdot (\mathbf{v} \times \mathbf{w}))$. If it is close to 0, then we know the vectors are nearly parallel. The dot product takes an additional 5 operations, or 14 total for our test. Recall that testing for parallel vectors using the dot product of nonnormalized vectors takes 18 operations; in this case, the cross product test is faster.

The cross product of two vectors is defined only for vectors in $\mathbb{R}^3$. However, in $\mathbb{R}^2$ we can define a similar operation on a single vector $\mathbf{v}$, called the *perpendicular*. This is represented as $\mathbf{v}^\perp$. The result of the perpendicular is the vector rotated 90 degrees. As with the cross product, we have two choices: in this case, counterclockwise or clockwise rotation. The standard definition is to rotate counterclockwise (Figure 2.18), so if $\mathbf{v} = (x, y)$, $\mathbf{v}^\perp = (-y, x)$.

The perpendicular has similar properties to the cross product. First, it produces a vector orthogonal to the original. Also, when used in combination with the dot product in $\mathbb{R}^2$ (also known as the *perpendicular dot product*),

$$\mathbf{v}^\perp \cdot \mathbf{w} = \|\mathbf{v}\| \|\mathbf{w}\| \sin\theta$$

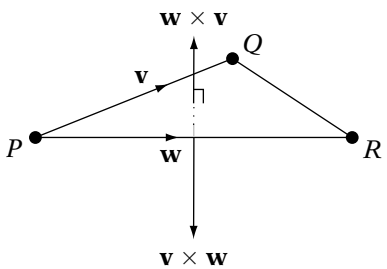


FIGURE 2.17 Computing normal for triangle.

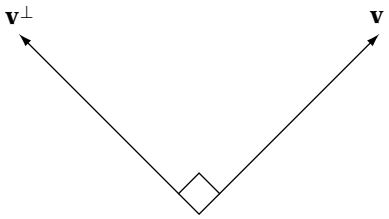


Figure 2.18 Perpendicular vector.

where θ is the *signed* angle between $\mathbf{v}$ and $\mathbf{w}$. That is, if the shortest rotation to get from $\mathbf{v}$ to $\mathbf{w}$ is in a clockwise direction, then θ is negative. And similar to the cross product, the absolute value of the perpendicular dot product is equal to the area of a parallelogram bordered by the two vectors.

It is possible to take cross products in dimensions greater than three by using $n-1$ vectors to take an n-dimensional cross product, but in general they won't be useful to us.

Our `IvVector3` cross product method is

```
IvVector3
IvVector3::Cross( const IvVector3& other )
{
    return IvVector3( y*other.z - other.y*z,
                      z*other.x - other.z*x,
                      x*other.y - other.x*y );
}
```

2.2.9 Triple Products

In $\mathbb{R}^3$ there are two extensions of the two single operation products called *triple products*. The first is the *vector triple product*, which returns a vector and is computed as $\mathbf{u} \times (\mathbf{v} \times \mathbf{w})$.

A special case is $\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$ (Figure 2.19). Examining this, $\mathbf{v} \times \mathbf{w}$ is perpendicular both to $\mathbf{v}$ and $\mathbf{w}$. The result of $\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$ is a vector perpendicular to both $\mathbf{w}$ and $(\mathbf{v} \times \mathbf{w})$. Therefore, if we combine normalized versions of $\mathbf{w}$, $(\mathbf{v} \times \mathbf{w})$, and $\mathbf{w} \times (\mathbf{v} \times \mathbf{w})$, we have an orthonormal basis (all are perpendicular and of unit length). This can be more efficient than Gram-Schmidt for producing orthogonal vectors, but of course it only works in $\mathbb{R}^3$.

The second triple product is called the *scalar triple product*. It (naturally) returns a scalar value, and its formula is $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w})$. To understand this geometrically, suppose we treat these three vectors as the edges of a slanted box,

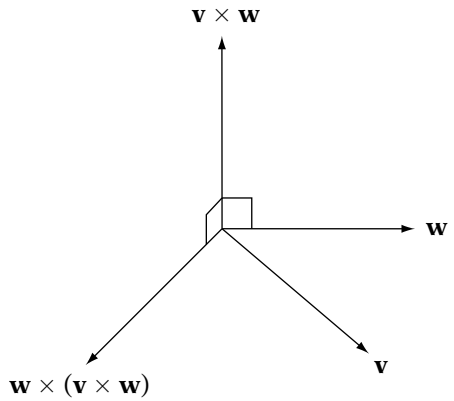


FIGURE 2.19 The vector triple product.

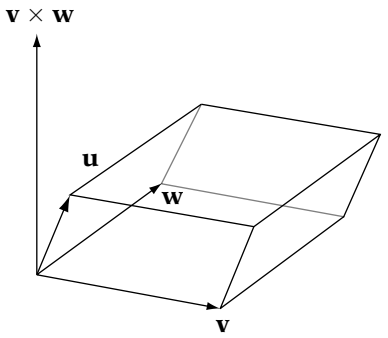


FIGURE 2.20 Scalar triple product equals volume of parallelopiped.

or parallelopiped (Figure 2.20). Then the area of the base equals $\|\mathbf{v} \times \mathbf{w}\|$, and $\|\mathbf{u}\| \cos\theta$ gives the height of the box. So,

$$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \|\mathbf{u}\| \|\mathbf{v} \times \mathbf{w}\| \cos\theta$$

or area times height equals the volume of the box.

In addition to computing volume, the scalar triple product can be used to test the direction of the angle between two vectors $\mathbf{v}$ and $\mathbf{w}$, relative to a third vector $\mathbf{u}$ that is linearly independent to both. If $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) > 0$, then the shortest rotation from $\mathbf{v}$ to $\mathbf{w}$ is in a counterclockwise direction (assuming our vectors are right-handed, as we will discuss shortly) around $\mathbf{u}$. Similarly, if $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) < 0$, the shortest rotation is in a relative clockwise direction.

For example, suppose we have a tank with current velocity $\mathbf{v}$ and desired direction $\mathbf{d}$ of travel. Our tank is oriented so that its current up direction points along a vector $\mathbf{u}$. We take the cross product $\mathbf{v} \times \mathbf{d}$ and dot it with $\mathbf{u}$. If

the result is positive, then we know that **d** lies to the left of **v** (counterclockwise rotation), and we turn left. Similarly, if the value is less than zero, then we know we must turn right to match **d** (Figures 2.21 and 2.22).

If we know that the tank is always oriented so that it lies on the xy plane, we can simplify this considerably. Vectors **v** and **d** will always have z values of 0, and **u** will always point in the same direction as the standard basis vector **k**. In this case, the result of $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{d})$ is equal to the z value of $\mathbf{v} \times \mathbf{d}$. So the problem simplifies to taking the cross product of **v** and **d** and checking the sign of the resulting z value to determine our turn direction.

Finally, we can use the scalar triple product to test whether ordered vectors in $\mathbb{R}^3$ are left-handed or right-handed. We can test this informally for our standard basis by using the right-hand rule. Take your right hand and point the thumb along **k** and your fingers along **i**. Now, rotating around your thumb, sweep your fingers counterclockwise into **j** (Figure 2.23). This

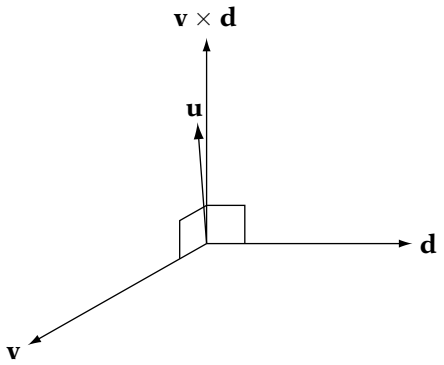


FIGURE 2.21 Scalar triple product indicates left turn.

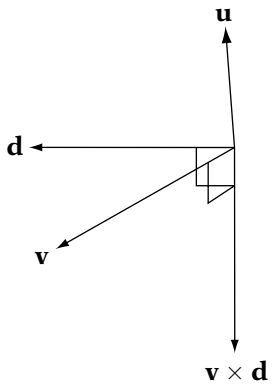


FIGURE 2.22 Scalar triple product indicates right turn.

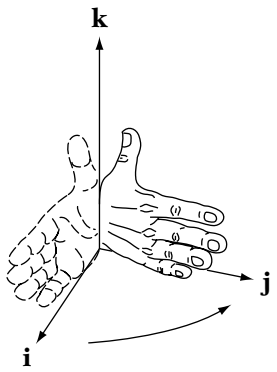


FIGURE 2.23 Right-handed rotation.

90-degree rotation of **i** into **j** shows that the basis is right-handed. We can do the same trick with the left hand rotating clockwise to show that a set of vectors is left-handed.

Formally, if we have three vectors $\{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2\}$, then they are right-handed if $\mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) > 0$, and left-handed if $\mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) < 0$. If $\mathbf{v}_0 \cdot (\mathbf{v}_1 \times \mathbf{v}_2) = 0$, we've got a problem — our vectors are linearly dependent.

While the scalar triple product only applies to vectors in $\mathbb{R}^3$, we can use the perpendicular dot product to test vectors in $\mathbb{R}^2$ for both turning direction and right- or left-handedness. For example, if we have two basis vectors $\{\mathbf{v}_0, \mathbf{v}_1\}$ in $\mathbb{R}^2$, then they are right-handed if $\mathbf{v}_0^{\perp} \cdot \mathbf{v}_1 > 0$ and left-handed if $\mathbf{v}_0^{\perp} \cdot \mathbf{v}_1 < 0$.

For vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ in $\mathbb{R}^3$, the following algebraic rules regarding the triple products apply:

1. $\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \cdot \mathbf{w})\mathbf{v} - (\mathbf{u} \cdot \mathbf{v})\mathbf{w}$.
2. $(\mathbf{u} \times \mathbf{v}) \times \mathbf{w} = (\mathbf{u} \cdot \mathbf{w})\mathbf{v} - (\mathbf{v} \cdot \mathbf{w})\mathbf{u}$.
3. $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \mathbf{w} \cdot (\mathbf{u} \times \mathbf{v}) = \mathbf{v} \cdot (\mathbf{w} \times \mathbf{u})$.

2.2.10 Real Vector Spaces

Up to this point, we have only been considering geometric vectors in 2D and 3D space and their representation using the standard Euclidean basis. However, there is an abstraction that can be useful to us. A *linear space*, or *vector space*, provides a formal means of encapsulating the concepts that we've just covered. This has a few advantages. First of all, since it is an abstraction, we can use it for manipulating higher-dimensional vectors than we might be able to conceive of geometrically. It also can be used for representing entities that we wouldn't normally consider as vectors but that follow the same algebraic

rules, which can be quite powerful. Finally, there are certain properties of vector spaces that will prove to be quite useful when we cover matrices and linear transformations.

To simplify our approach, we are going to concentrate on a subset of vector spaces known as *real vector spaces*, so called because their fundamental components are drawn from $\mathbb{R}$, the set of all real numbers. We usually say that such a vector space V is *over* $\mathbb{R}$. We also formally define an element of $\mathbb{R}$ in this case as a *scalar*.

So what is a real vector space? One example of a real vector space is simply $\mathbb{R}$. At first glance it may be difficult to see the correspondence between a real number and a vector, but as we'll see next, $\mathbb{R}$ does meet the criteria for a vector space.

We've already seen another vector space: $\mathbb{R}^2$. As mentioned, we can think of this as informally representing 2D space. Symbolically, this is represented by

$$\mathbb{R}^2 = \{(x, y) \mid x, y \in \mathbb{R}\}$$

In this context, the symbol | means "such that" and the symbol $\in$ means "is a member of." So we read this as "The set of all possible pairs (x, y), such that x and y are members of the set of real numbers." And as before, this is a set of ordered pairs; $(1.0, -0.5)$ is a different member of the set from $(-0.5, 1.0)$.

We define $\mathbb{R}^3$ and $\mathbb{R}^4$ similarly as follows:

$$\mathbb{R}^3 = \{(x, y, z) \mid x, y, z \in \mathbb{R}\}$$

$$\mathbb{R}^4 = \{(w, x, y, z) \mid w, x, y, z \in \mathbb{R}\}$$

Like $\mathbb{R}^2$, these are ordered lists, where two members with the same values but differing orders are not the same. As we've seen, $\mathbb{R}^3$ informally represents positions in 3D space. Correspondingly, $\mathbb{R}^4$ can be thought of as representing 4D space, which is difficult to visualize spatially[2] (hence our need for an abstract representation), but is extremely useful for certain computer graphics concepts.

We can extend our definitions to $\mathbb{R}^n$, a generalized n-dimensional space over $\mathbb{R}$:

$$\mathbb{R}^n = \{(x_0, \ldots, x_{n-1}) \mid x_0, \ldots, x_{n-1} \in \mathbb{R}\}$$

The members of $\mathbb{R}^n$ are referred to as an n-tuple.

Up until now we've been casually referring to these real-number spaces as vector spaces. For them to be proper vector spaces and not just organized lists

2. Unless you are one of a particularly gifted pair of children [87].

of numbers, we need to define two specific operations on the elements that follow certain algebraic rules. The two operations should be familiar from our discussion of geometric vectors: They are addition and scalar multiplication. We'll define these operations so that the vector space V has *closure* with respect to them; that is,

1. For any $\mathbf{u}$ and $\mathbf{v}$ in V, $\mathbf{u} + \mathbf{v}$ is in V (additive closure).
2. For any a in $\mathbb{R}$ and $\mathbf{v}$ in V, $a\mathbf{v}$ is in V (multiplicative closure).

So formally, we define a real vector space as a set V over $\mathbb{R}$ with closure with respect to addition and scalar multiplication on its elements, where the following properties hold:

For all $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$, $\mathbf{0}$ in V and all a, b in $\mathbb{R}$:

1. $\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$ (commutative property).
2. $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ (associative property).
3. There exists an element $\mathbf{0}$ such that $\mathbf{v} + \mathbf{0} = \mathbf{v}$ (additive identity).
4. For every $\mathbf{v}$, there is an element $-\mathbf{v}$ such that $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$ (additive inverse).
5. $(ab)\mathbf{v} = a(b\mathbf{v})$ (associative property).
6. $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$ (distributive property).
7. $a(\mathbf{v} + \mathbf{w}) = a\mathbf{v} + a\mathbf{w}$ (distributive property).
8. $1 \cdot \mathbf{v} = \mathbf{v}$ (multiplicative identity).

These are exactly the properties we stated previously for vector addition and scalar multiplication.

As an example, we can use our previous definition of addition in $\mathbb{R}^2$:

$$(x_0, y_0) + (x_1, y_1) = (x_0 + x_1, y_0 + y_1)$$

and scalar multiplication:

$$a(x_0, y_0) = (ax_0, ay_0)$$

Using these definitions and the preceding algebraic axioms, it can be shown that $\mathbb{R}^2$ is a vector space. Similar operations can be defined for $\mathbb{R}^3$ and $\mathbb{R}^4$, as

well as for $\mathbb{R}$ itself. Generalized over $\mathbb{R}^n$, we have

$$\begin{aligned}\mathbf{u} + \mathbf{v} &= (u_0, \ldots, u_{n-1}) + (v_0, \ldots, v_{n-1}) \\ &= (u_0 + v_0, \ldots, u_{n-1} + v_{n-1})\end{aligned}$$

and

$$\begin{aligned}a\mathbf{v} &= a(v_0, \ldots, v_{n-1}) \\ &= (av_0, \ldots, av_{n-1})\end{aligned}$$

Now suppose we have a subset W of a vector space V. We call W a *subspace* if it is itself a vector space when using the same definition for addition and multiplication operations. In order to show that a given subset W is a vector space, we only need to show that closure under addition and scalar multiplication holds; the rest of the properties are satisfied because W is a subset of V. For example, the subset of all vectors in $\mathbb{R}^3$ with $z = 0$ is a subspace, since

$$\begin{aligned}(x_0, y_0, 0) + (x_1, y_1, 0) &= (x_0 + x_1, y_0 + y_1, 0) \\ a(x_0, y_0, 0) &= (ax_0, ay_0, 0)\end{aligned}$$

The resulting vectors still lie in the subspace $\mathbb{R}^3$ with $z = 0$.

Note that any subspace must contain $\mathbf{0}$ in order to meet the conditions for a vector space. So the subset of all vectors in $\mathbb{R}^3$ with $z = 1$ is not a subspace since $\mathbf{0}$ cannot be represented. And while $\mathbb{R}^2$ is not a subspace of $\mathbb{R}^3$ (since the former is a set of pairs and the latter a set of triples), it can be embedded in a subspace of $\mathbb{R}^3$ by a mapping, for example, $(x, y) \rightarrow (x, y, 0)$.

It is important to understand that — despite the name — a vector space does not necessarily have to be made up of geometric vectors. What we have described is a series of sets of ordered lists, possibly with no relation to a geometric construct. As we have seen, they *can* be related to the geometry, but the term *vector*, when used in describing members of vector spaces, is an abstract concept. As long as a set of elements can be shown to have the preceding arithmetic properties, we define it as a vector space and any element of a vector space as a vector. It is perhaps more correct to say that the geometric representations of 2D and 3D vectors that we use are visualizations that help us better understand the abstract nature of $\mathbb{R}^2$ and $\mathbb{R}^3$, rather than the other way around.

2.2.11 Basis Vectors

Now suppose that for a given vector space V, we can find a set β of n linearly independent vectors in V that span V. With this we can formally define β as

a *basis* for V, and each element of β as a *basis vector*. So far we've shown only the standard Euclidean basis, but other bases are possible for a given vector space, and they will always have the same number of elements. We formally define a vector space's *dimension* as equal to the number of basis vectors required to span it. So, for example, any basis for $\mathbb{R}^3$ will contain three basis vectors, and so it is (as we'd expect) a 3D space. Note that while the standard Euclidean basis is orthonormal, this is not necessary. Basis vectors can have nonunit length and be nonorthogonal. All that is required is that they be linearly independent.

As mentioned, among the many bases for a vector space, we define one as the *standard basis*. In general this is represented as $\{\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}\}$, where

$$\begin{aligned} \mathbf{e}_0 &= (1, 0, \ldots, 0) \\ \mathbf{e}_1 &= (0, 1, \ldots, 0) \\ &\vdots \\ \mathbf{e}_{n-1} &= (0, 0, \ldots, 1) \end{aligned}$$

One property of a basis β is that for every vector $\mathbf{v}$ in V, there is a unique linear combination of the vectors in β that equal $\mathbf{v}$. So, using a general basis $\beta = \{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}\}$, there is only one list of coefficients $a_0, \ldots, a_{n-1}$ such that

$$\mathbf{v} = a_0\mathbf{b}_0 + a_1\mathbf{b}_1 + \cdots + a_{n-1}\mathbf{b}_{n-1}$$

This formally explains why, instead of using the full equation to represent $\mathbf{v}$, we can abbreviate it by using only the coefficients $a_0, \ldots, a_{n-1}$ and store them in an ordered n-tuple as $(a_0, \ldots, a_{n-1})$. Note that the coefficient values will be dependent on which basis we're using and will almost certainly be different from basis to basis. The ordering of the basis vectors is important: A different ordering will not necessarily generate the same coefficients for a given vector. For most cases, though, we'll be assuming the standard basis, as we did above.

2.3 Points

Now that we have covered vectors and vector operations in some detail, we turn our attention to a related entity: the point. While the reader probably has some intuitive notion of what a point is, in this section we'll provide a mathematical representation and discuss the relationship between vectors and points. We'll also discuss some special operations that can be performed on points and alternatives to the standard Cartesian coordinate system.

Within this section it is also assumed that the reader has some general sense of what lines and planes are. More information on these topics follows in subsequent sections.

2.3.1 Points as Geometry

Everyone who has been through a first-year geometry course should be familiar with the notion of a *point*. Euclid describes the point in his work *Elements* [33] as "that which has no part." Points have also been presented as the cross-section of a line, or the intersection of two lines. A less vague but still not satisfactory definition is to describe them as an infinitely small entity that has only the property of location. In games we use points for two primary purposes: to represent the position of game objects and as the basic building block of their geometric representation. Points are represented graphically by a dot.

Euclid did not present a means for representing position numerically, although later Greek mathematicians used latitude, longitude, and altitude. The primary system we use now—Cartesian coordinates—was originally published by Rene Descartes in his 1637 work *La geometrie* [24] and further revised by Newton and Leibniz.

In this system we measure a point's location relative to a special, anchored point, called the *origin*, which is represented by the letter O. In $\mathbb{R}^2$ we informally define two perpendicular real-number lines or axes—known as the x- and y-axes—that pass through the origin. We indicate the location of a point P by a pair (x, y) in $\mathbb{R}^2$, where x is the distance from the point to the y-axis, and y is the distance from the point to the x-axis. Another way to think of it is that we count x units along the x-axis and then y units up parallel to the y-axis to reach the point's location. This combination of origin and axes is called the *Cartesian coordinate system* (Figure 2.24).

For $\mathbb{R}^3$ three perpendicular coordinate axes—x, y, and z—intersect at the origin. There are corresponding coordinate planes xy, yz, and xz that also intersect at the origin. Take the room you're sitting in as our space, with one corner of the room as the origin, and think of the walls and floor as the three coordinate planes (assume they extend infinitely). The edges where the walls and floor join together correspond to the axes. We can think of a 3D position as being a real-number triple (x, y, z) corresponding to the distance of the point to the three planes, or counting along each axis as before.

In Figure 2.25 you can see an example of a 3D coordinate system. Here the axis pointing up is called the z-axis, the one to the side is the y-axis, and the one aimed slightly out of the page is the x-axis. Another system that is commonly used in graphics books has the y-axis pointing up, the x-axis to the right, and the z-axis out of the page (Figure 2.26). Some graphics developers

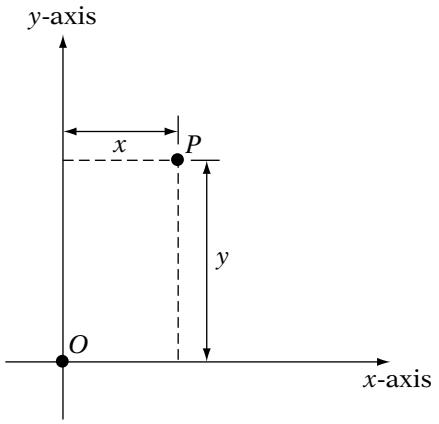


FIGURE 2.24 Two-dimensional Cartesian coordinate system.

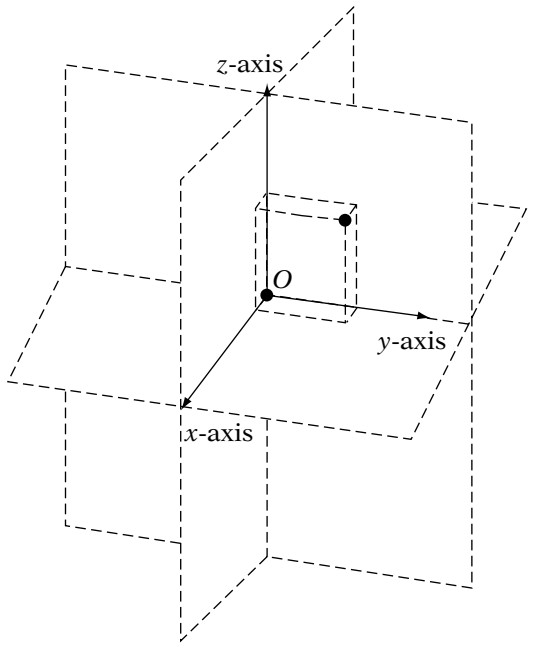


FIGURE 2.25 Three-dimensional Cartesian coordinate system.

favor this because the *x*- and *y*-axes match the relative axes of the 2D screen, but most of the time we'll be using the former convention for this book.

Both of the 3D coordinate systems we have described are right-handed. As before, we can test this via the right-hand rule. This time point your thumb along the *z*-axis, your fingers along the *x*-axis, and rotate counterclockwise

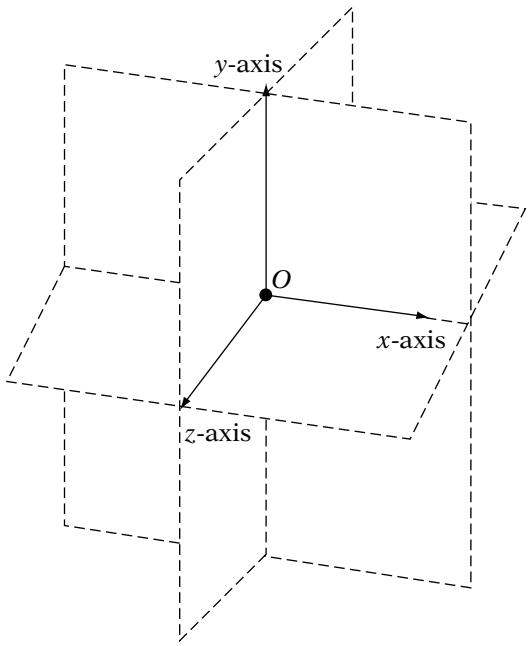


FIGURE 2.26 Alternate 3D Cartesian coordinate system.

into the y-axis. As with left-handed bases, we can have left-handed coordinate systems (and will be using them later in this book), but the majority of our work will be done in a right-handed coordinate system because of convention.

2.3.2 AFFINE SPACES

We can provide a more formal definition of coordinate systems based on what we already know of vectors and vector spaces. Before we can do so, though, we need to define the relationship between vectors and points. Points can be related to vectors by means of an *affine space*. An affine space consists of a set of points W and a vector space V. The relation between the points and vectors is defined using the following two operations: For every pair of points P and Q in W, there is a unique vector $\mathbf{v}$ in V such that

$$\mathbf{v} = Q - P$$

Correspondingly, for every point P in W and every vector $\mathbf{v}$ in V, there is a unique point Q such that

$$Q = P + \mathbf{v} \tag{2.7}$$

This relationship can be seen in Figure 2.27. We can think of the vector $\mathbf{v}$ as acting as a displacement between the two points P and Q. To determine the displacement between two points, we subtract one from another. To displace a point, we add a vector to it and that gives us a new point.

We can define a fixed-point O in W, known as the *origin*. Then using equation 2.7, we can represent any point P in W as

$$P = O + \mathbf{v}$$

or, expanding our vector using n basis vectors that span V:

$$P = O + a_0\mathbf{v}_0 + a_1\mathbf{v}_1 + \cdots + a_{n-1}\mathbf{v}_{n-1} \tag{2.8}$$

Using this, we can represent our point using an n-tuple $(a_0, \ldots, a_{n-1})$ just as we do for vectors. The combination of the origin O and our basis vectors $(\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})$ is known as a *coordinate frame*.

Note that we can use any point in W as our origin and—for an n-dimensional affine space—any n linearly independent vectors as our basis. Unlike the Cartesian axes, this basis does not have to be orthonormal, but using an orthonormal basis (as with vectors) does make matching our physical geometry with our abstract representation more straightforward. Because of this, we will work with the standard origin $(0, 0, \ldots, 0)$, and the standard basis $\{(1, 0, \ldots, 0), (0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$. This is known as the *Cartesian frame*.

In $\mathbb{R}^3$ our Cartesian frame will be the origin $O = (0, 0, 0)$ and the standard ordered basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ as before. Our basis vectors will lie along the x-, y-, and z-axes, respectively. By using this system, we can use the same triple (x, y, z) to represent a point and the corresponding vector from the origin to the point (Figure 2.28).

To compute the distance between two points we use the length of the vector that is their difference. So, if we have two points $P_0 = (x_0, y_0, z_0)$ and $P_1 = (x_1, y_1, z_1)$ in $\mathbb{R}^3$, the difference is

$$\mathbf{v} = P_1 - P_0 = (x_1 - x_0, y_1 - y_0, z_1 - z_0)$$

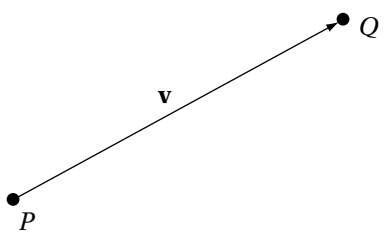


FIGURE 2.27 Affine relationship between points and vectors.

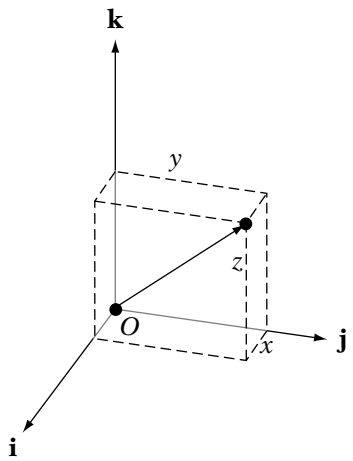


FIGURE 2.28 Relationship between points and vectors in Cartesian affine frame.

and the distance between them is

$$\text{dist}(P_1, P_0) = \|v\| = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

This is also known as the *Euclidean distance*. In the $\mathbb{R}^3$ Cartesian frame, the distance between a point $P = (x, y, z)$ and the origin is

$$\text{dist}(P, O) = \sqrt{x^2 + y^2 + z^2}$$

2.3.3 AFFINE COMBINATIONS

So far the only operation that we've defined on points alone is subtraction, which results in a vector. However, there is a limited addition operation that we can perform on points that gives us a point as a result. It is known as an *affine combination*, and has the form

$$P = a_0 P_0 + a_1 P_1 + \cdots + a_k P_k \tag{2.9}$$

where

$$a_0 + a_1 + \cdots + a_k = 1 \tag{2.10}$$

So, an affine combination of points is like a linear combination of vectors, with the added restriction that all the coefficients need to add up to 1. We can

show why this restriction allows us to perform this operation by rewriting equation 2.10 as

$$a_0 = 1 - a_1 - \cdots - a_k$$

and substituting into equation 2.9 to get

$$\begin{aligned} P &= (1 - a_1 - \cdots - a_k)P_0 + a_1 P_1 + \cdots + a_k P_k \\ &= P_0 + a_1(P_1 - P_0) + \cdots + a_k(P_k - P_0) \end{aligned} \tag{2.11}$$

If we set $\mathbf{u}_1 = (P_1 - P_0)$, $\mathbf{u}_2 = (P_2 - P_0)$, and so on, we can rewrite this as

$$P = P_0 + a_1 \mathbf{u}_1 + a_2 \mathbf{u}_2 + \cdots + a_k \mathbf{u}_k$$

So, by restricting our coefficients in this manner, it allows us to rewrite the affine combination as a point plus a linear combination of vectors, a perfectly legal operation.

Looking back at our coordinate frame equation 2.8, we can see that it too is an affine combination. Just as we use the coefficients in a linear combination of basis vectors to represent a general vector, we can use the coefficients of an affine combination of origin and basis vectors to represent a general point.

An affine combination spans an affine space, just as a linear combination spans a vector space. If the vectors in equation 2.11 are linearly independent, we can represent any point in the spanned affine space using the coefficients of the affine combination, just as we did before with vectors. In this case, we say that the points $P_0, P_1, \ldots, P_k$ are *affinely independent*, and the ordered points are called a *simplex*. The coefficients are called *barycentric coordinates*. For example, we can create an affine combination of a simplex made of three affinely independent points P_0, P_1, and P_2. The affine space spanned by the affine combination $a_0 P_0 + a_1 P_1 + a_2 P_2$ is a plane, and any point in the plane can be specified by the coordinates (a_0, a_1, a_2).

We can further restrict the set of points spanned by the affine combination by considering properties of convex sets. A *convex set* of points is defined such that a line drawn between any pair of points in the set remains within the set (Figure 2.29). The *convex hull* of a set of points is the smallest convex set that includes all the points. If we restrict our coefficients $(a_0, \ldots, a_{n-1})$ such that $0 \leq a_0, \ldots, a_{n-1} \leq 1$, then we have a *convex combination*, and the span of the convex combination is the convex hull of the points. For example, the convex combination of three affinely independent points spans a triangle. We will discuss the usefulness of this in more detail when we cover triangles in Section 2.6.

If the barycentric coordinates in a convex combination of n points are all $1/n$, then the point produced is called the *centroid*, which is the mean of a set of points.

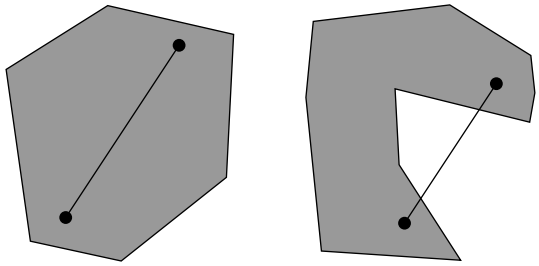


FIGURE 2.29 Convex versus nonconvex set of points.

2.3.4 POINT IMPLEMENTATION

SOURCE CODE
LIBRARY
lvMath
FILENAME
lvVector3

Using the Cartesian frame and standard basis in $\mathbb{R}^3$, the x, y, and z values of a point P in $\mathbb{R}^3$ match the x, y, and z values of the corresponding vector $P - O$, where O is the origin of the frame. This also means that we can use one class to represent both, since one can be easily converted to the other. Because of this, many math libraries don't even bother implementing a point class and just treat points as vectors.

Other libraries indicate the difference by treating them both as 4-tuples and indicate a point as $(x, y, z, 1)$ and a vector as $(x, y, z, 0)$. In this system if we subtract a point from a point, we automatically get a vector:

$$(x_0, y_0, z_0, 1) - (x_1, y_1, z_1, 1) = (x_0 - x_1, y_0 - y_1, z_0 - z_1, 0)$$

Similarly, a point plus a vector produces a point:

$$(x_0, y_0, z_0, 1) + (x_1, y_1, z_1, 0) = (x_0 + x_1, y_0 + y_1, z_0 + z_1, 1)$$

Even affine combinations give the expected results:

$$\sum_{i=0}^{n-1} a_i(x_i, y_i, z_i, 1) = \left(\sum_{i=0}^{n-1} a_i x_i, \sum_{i=0}^{n-1} a_i y_i, \sum_{i=0}^{n-1} a_i z_i, \sum_{i=0}^{n-1} a_i \right)$$

$$= \left(\sum_{i=0}^{n-1} a_i x_i, \sum_{i=0}^{n-1} a_i y_i, \sum_{i=0}^{n-1} a_i z_i, 1 \right)$$

OpenGL uses this form when specifying the difference between a point light, which casts light rays in all directions from a given position, and

a directional light, which only casts light rays in one direction. Both are specified by a single call:

```
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

If the final value of `light_position` is 0, then it is treated as a directional light; otherwise, it is treated as a point light.

In our case, we will not be using a separate class for points. There would be a certain amount of code duplication, since the `IvPoint3` class would end up being very similar to the `IvVector3` class. Also to be considered is the performance cost of converting points to vectors and back again. Further, to maintain type correctness we may end up distorting equations unnecessarily; this obfuscates the code and can lead to a loss in performance as well. Finally, most production game engines don't make the distinction, and we wish to remain compatible with the overall state of the industry.

Despite not making the distinction in the class structure, it is important to remember that points and vectors are not the same. One has direction and length and the other position, so not all operations apply to both. For example, we can add two vectors together to get a new vector. As we've seen, adding two points together is only allowed in certain circumstances. So, while we will be using a single class, we will be maintaining mathematical correctness in the text and writing the code to reflect this.

As mentioned, most of what we need for points is already in the `IvVector3` class. The only additional code we'll have to implement is for distance and distance squared operations:

```
float
Distance( const IvVector3& point1,
          const IvVector3& point2 )
{
    float x = point1.x - point2.x;
    float y = point1.y - point2.y;
    float z = point1.z - point2.z;

    return IvSqrt( x*x + y*y + z*z );
}

float
DistanceSquared( const IvVector3& point1,
                 const IvVector3& point2 )
{
    float x = point1.x - point2.x;
```

```
        float y = point1.y - point2.y;
        float z = point1.z - point2.z;

        return ( x*x + y*y + z*z );
    }
```

2.3.5 Polar and Spherical Coordinates

Cartesian coordinates are not the only way of measuring location. We've already mentioned latitude, longitude, and altitude, and there are other, related systems. Take a point P in $\mathbb{R}^2$ and compute the vector $\mathbf{v} = P - O$. We can specify the location of P using the distance r from P to the origin, which is the length of $\mathbf{v}$, and the angle θ between $\mathbf{v}$ and the positive x-axis, where $\theta > 0$ corresponds to a counterclockwise rotation from the axis. The components (r, θ) are known as *polar coordinates*.

It is easy to convert from polar to Cartesian coordinates. We begin by forming a right triangle using the x-axis, a line from P to O, and the perpendicular from P to the x-axis (Figure 2.30). The hypotenuse has the length r and is θ degrees from the x-axis. Using simple trigonometry, the lengths of the other two sides of the triangle x and y can be computed as

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned} \tag{2.12}$$

From Cartesian to polar coordinates, we reverse the process. It's easy enough to generate r by computing the distance between P and O. Finding θ is not as straightforward. The naive approach is to solve equation 2.12

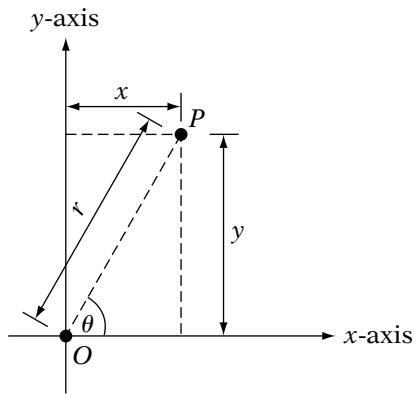


Figure 2.30 Relationship between polar and Cartesian coordinates.

for θ, which gives us $\theta = \arccos(x/r)$. However, the `acos()` function under C++ only returns an angle in the range of $[0, \pi)$, so we've lost the sign of the angle. Since

$$\begin{aligned}\frac{y}{x} &= \frac{r\sin\theta}{r\cos\theta} \\ &= \frac{\sin\theta}{\cos\theta} \\ &= \tan\theta\end{aligned}$$

an alternate choice would be $\arctan(y/x)$, but this doesn't handle the case when $x = 0$. To manage this, C++ provides a library function called `atan2()`, which takes y and x as separate arguments and computes $\arctan(y/x)$. It has no problems with division by 0 and maintains the signed angle with a range of $[-\pi, \pi]$. We'll represent the use of this function in our equations as $\arctan 2(y, x)$. The final result is

$$\begin{aligned}r &= \sqrt{x^2 + y^2} \\ \theta &= \arctan 2(y, x)\end{aligned}$$

If r is 0, θ may be set arbitrarily.

The system that extends this to three dimensions is called *spherical coordinates*. In this system we call the distance from the point to the origin ρ instead of r. We create a sphere of radius ρ centered on the origin and define where the point lies on the sphere by two angles, ϕ and θ. If we take a vector $\mathbf{v}$ from the origin to the point and project it down to the xy plane, θ is the angle between the x-axis and rotating counterclockwise around z. The other quantity, ϕ, measures the angle between $\mathbf{v}$ and the z-axis. The three values, ρ, ϕ, and θ, represent the location of our point (Figure 2.31).

Spherical coordinates can be converted to Cartesian coordinates as follows. Begin by building a right triangle as before, except with its hypotenuse along ρ and base along the z-axis (Figure 2.32). The length z is then $\rho\cos\phi$. To compute x and y, we project the vector $\mathbf{v}$ down onto the xy plane, and then use polar coordinates. The length r of the projected vector $\mathbf{v}'$ is $\rho\sin\phi$, so we have

$$x = \rho\sin\phi\cos\theta \tag{2.13}$$

$$y = \rho\sin\phi\sin\theta \tag{2.14}$$

$$z = \rho\cos\phi \tag{2.15}$$

To convert from Cartesian to spherical coordinates, we begin by computing ρ, which again is the distance from the point to the origin. To find ϕ, we

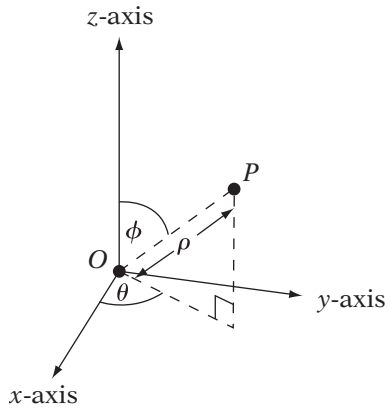


FIGURE 2.31 Spherical coordinates.

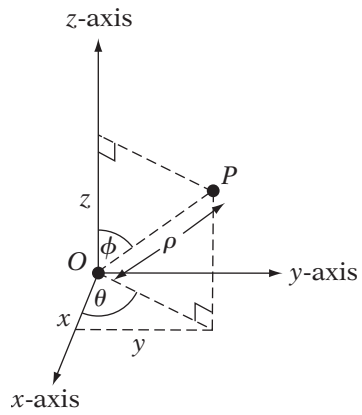


FIGURE 2.32 Relationship between spherical and Cartesian coordinates.

need to find the value of $\rho \sin \phi$. This is equal to the projected xy length r since

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ &= \sqrt{(\rho \sin \phi \cos \theta)^2 + (\rho \sin \phi \sin \theta)^2} \\ &= \sqrt{(\rho \sin \phi)^2 (\cos^2 \theta + \sin^2 \theta)} \\ &= \rho \sin \phi \end{aligned}$$

And since, as with polar coordinates,

$$\frac{r}{z} = \frac{\rho \sin \phi}{\rho \cos \phi}$$
$$= \tan \phi$$

we can compute $\phi = \arctan 2(r, z)$. Similarly, $\theta = \arctan 2(y, x)$. Summarizing:

$$\rho = \sqrt{x^2 + y^2 + z^2}$$
$$\phi = \arctan 2\left(\sqrt{x^2 + y^2}, z\right)$$
$$\theta = \arctan 2(y, x)$$

2.4 Lines

2.4.1 Definition

As with the point, a *line* as a geometric concept should be familiar. Euclid [33] defines a line as "breadthless length" and a *straight line* as that "which lies evenly with the points on itself." A straight line also has been referred to as the shortest distance between two points, although in non-Euclidean geometry this is not necessarily true.

From first-year algebra, we know that a line in $\mathbb{R}^2$ is represented by the formula

$$y = mx + b \tag{2.16}$$

where m is the slope of the line (it describes how y changes with each step of x), and b is the coordinate location where the line crosses the y-axis (called the y-intercept). In this case, x varies over all values and y is represented in terms of x. This general form works for all lines in $\mathbb{R}^2$ except for those that are vertical, since in that case the slope is infinite and the y-intercept is either nonexistent or is all values along the y-axis.

Equation 2.16 has a few problems. First of all, as mentioned, we can't easily represent a vertical line — it has infinite slope. And, it isn't obvious how to transform this equation into one useful for three dimensions. We will need a different representation.

2.4.2 Parameterized Lines

One possible representation is known as a *parametric equation*. Instead of representing the line as a single equation with a number of variables, each coordinate value is calculated by a separate function. This allows us to use one form for a line that is generalizable across all dimensions. As an example, we will take equation 2.16 and parameterize it.

To compute the parametric equation for a line, we need two points on our line. We can take the y-intercept $(0, b)$ as one of our points, and then take one step in the positive x direction, or $(1, m + b)$, to get the other. Subtracting point 1 from point 2, we get a 2D vector $\mathbf{d} = (1, m)$, which is oriented in the same direction as the line (Figure 2.33). If we take this vector and add all the possible scalar multiples of it to the starting point $(0, b)$, then the points generated will lie along the line. We can express this in one of the following forms:

$$L(t) = P_0 + t(P_1 - P_0) \tag{2.17}$$

$$= (1 - t)P_0 + tP_1 \tag{2.18}$$

$$= P_0 + t\mathbf{d} \tag{2.19}$$

The variable t in this case is called a *parameter*.

We started with a 2D example, but the formulas we just derived work beyond two dimensions. As long as we have two points, we can just substitute them into the preceding equations to represent a line. More formally, if we examine equation 2.17, we see it matches equation 2.11. The affine combination of two unequal or noncoincident points span a line. Equation 2.19 makes this even clearer. If we think of P_0 as our origin and $\mathbf{d}$ as a basis vector, they span a 1D affine space, which is the line.

Since our line is spanned by an affine combination of our two points, the logical next question is: What is spanned by the convex combination? The convex combination requires that t and $(1 - t)$ lie between 0 and 1, which holds only if t lies in the interval $[0, 1]$. Clamping t to this range gives us a *line segment* (Figure 2.34). The edges of polygons are line segments, and we'll also be using line segments when we talk about bounding objects and collision detection.

If we clamp t to only one end of the range, usually specifying that $t \geq 0$, then we end up with a *ray* (Figure 2.35) that starts at P_0 and extends infinitely

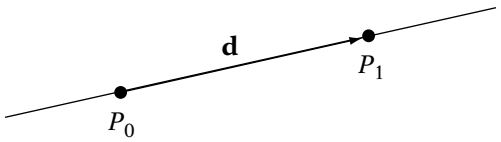


Figure 2.33 Line.

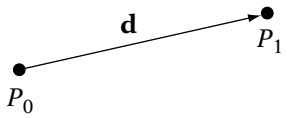


FIGURE 2.34 Line segment.

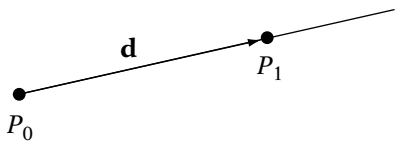


FIGURE 2.35 Ray.

along the line in the direction of **d**. Rays are useful for intersection and visibility tests. For example, P_0 may represent the position of a camera, and **d** is the viewing direction.

SOURCE CODE
LIBRARY
IvMath
FILENAME
IvLine3
IvLineSegment3
IvRay3

In code we'll be representing our lines, rays, and line segments as a point on the line P and a vector **d**; so for example, the class definition for a line in $\mathbb{R}^3$ is

```
class IvLine3
{
public:
  IvLine3( const IvVector3& direction, const IvVector3& origin );

  IvVector3 mDirection;
  IvPoint3  mOrigin;
};
```

2.4.3 Generalized Line Equation

There is another formulation of our 2D line that can be useful. Let's start by writing out the equations for both x and y in terms of t:

$$x = P_x + td_x$$
$$y = P_y + td_y$$

Solving for t in terms of x, we have

$$t = \frac{(x - P_x)}{d_x}$$

Substituting this into the y equation, we get

$$y = d_y \frac{(x - P_x)}{d_x} + P_y$$

We can rewrite this as

$$\begin{aligned} 0 &= \frac{(y - P_y)}{d_y} - \frac{(x - P_x)}{d_x} \\ &= (-d_y)x + (d_x)y + (d_y P_x - d_x P_y) \\ &= ax + by + c \end{aligned} \tag{2.20}$$

where

$$\begin{aligned} a &= -d_y \\ b &= d_x \\ c &= d_y P_x - d_x P_y = -aP_x - bP_y \end{aligned}$$

We can think of a and b as the components of a 2D vector $\mathbf{n}$, which is perpendicular to the direction vector $\mathbf{d}$, and so is orthogonal to the direction of the line (Figure 2.36). This gives us a way of testing where a 2D point lies relative to a 2D line. If we substitute the coordinates of the point into the x and y values of the equation, then a value of 0 indicates it's on the line, a positive value indicates that it's on the side where the vector is pointing, and a negative value indicates that it's on the opposite side. If we normalize our vector, we can use the value returned by the line equation to indicate the distance from the point to the line.

To see why this is so, suppose we have a test point Q. We begin by constructing the vector between Q and our line point P, or $Q - P$. There are two possibilities. If Q lies on the side of the line where $\mathbf{n}$ is pointing, then the distance between Q and the line is

$$d = \|Q - P\| \cos\theta$$

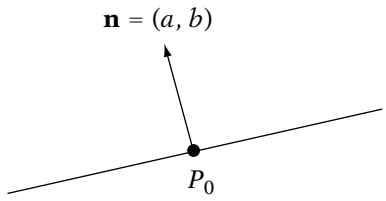


Figure 2.36 Normal form of 2D line.

where θ is the angle between $\mathbf{n}$ and $Q - P$. But since $\mathbf{n} \cdot (Q - P) = \|\mathbf{n}\| \|Q - P\| \cos\theta$, we can rewrite this as

$$d = \frac{\mathbf{n} \cdot (Q - P)}{\|\mathbf{n}\|}$$

If Q is lying on the opposite side of the line, then we take the dot product with the negative of $\mathbf{n}$, so

$$\begin{aligned} d &= \frac{-\mathbf{n} \cdot (Q - P)}{\|-\mathbf{n}\|} \\ &= -\frac{\mathbf{n} \cdot (Q - P)}{\|\mathbf{n}\|} \end{aligned}$$

Since d is always positive, we can just take the absolute value of $\mathbf{n} \cdot (Q - P)$ to get

$$d = \frac{|\mathbf{n} \cdot (Q - P)|}{\|\mathbf{n}\|} \tag{2.21}$$

If we know that $\mathbf{n}$ is normalized, we can drop the denominator. If $Q = (x, y)$ and (as we've stated) $\mathbf{n} = (a, b)$, we can expand our values to get

$$\begin{aligned} d &= a(x - P_x) + b(y - P_y) \\ &= ax + by - aP_x - bP_y \\ &= ax + by + c \end{aligned}$$

If our $\mathbf{n}$ is not normalized, then we need to remember to divide by $\|\mathbf{n}\|$ to get the correct distance.

2.4.4 Collinear Points

Three or more points are said to be *collinear* if they all lie on a line. Another way to think of this is that despite there being more than two points, the affine space that they span is only one dimensional.

To determine whether three points P_0, P_1, and P_2 are collinear, we take the cross product of $P_1 - P_0$ and $P_2 - P_0$ and test whether the result is close to the zero vector. This is equivalent to testing whether basis vectors for the affine space are parallel.

2.5 Planes

Euclid [33] defines a surface as "that which has length and breadth only," and a plane surface, or just a *plane*, as "a surface which lies evenly with the straight lines on itself." Another way of thinking of this is that a plane is created by taking a straight line and sweeping each point on it along a second straight line. It is a flat, limitless, infinitely thin surface.

2.5.1 Parameterized Planes

As with lines, we can express a plane algebraically in a number of ways. The first follows from our parameterized line. From basic geometry we know that two noncoincident points form a line and three noncollinear points form a plane. So, if we can parameterize a line as an affine combination of two points, then it makes sense that we can parameterize a plane as an affine combination of three points P_0, P_1, and P_2, or

$$P(s,t) = (1-s-t)P_0 + sP_1 + tP_2$$

Alternatively, we can represent this as an origin point plus the linear combination of two vectors:

$$\begin{aligned} P(s,t) &= P_0 + s(P_1 - P_0) + t(P_2 - P_0) \\ &= P_0 + s\mathbf{u} + t\mathbf{v} \end{aligned}$$

As with the parameterized line equation, if our points are of higher dimension, we can create planes in higher dimensions from them. However, in most cases our planes will be firmly entrenched in $\mathbb{R}^3$.

2.5.2 Generalized Plane Equation

We can define an alternate representation for a plane in $\mathbb{R}^3$, just as we did for a line in $\mathbb{R}^2$. In this form a plane is defined as the set of points perpendicular to a normal vector $\mathbf{n} = (a, b, c)$ that also contains the point $P_0 = (x_0, y_0, z_0)$ as shown in Figure 2.37. If a point P lies on the plane, then the vector $\mathbf{v} = P - P_0$ also lies on the plane. For $\mathbf{v}$ and $\mathbf{n}$ to be orthogonal, then $\mathbf{n} \cdot \mathbf{v} = 0$. Expanding this gives us the *normal-point* form of the plane equation, or

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

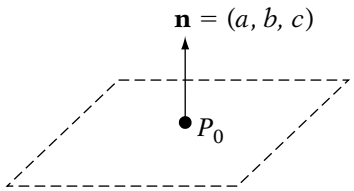


FIGURE 2.37 Normal form of plane.

We can pull all the constants into one term to get

$$\begin{aligned} 0 &= ax + by + cz - (ax_0 + by_0 + cz_0) \\ &= ax + by + cz + d \end{aligned}$$

So, extending equation 2.20 to three dimensions gives us the equation for a plane in $\mathbb{R}^3$.

This is the *generalized plane equation*. As with the generalized line equation, this equation can be used to test where a point lies relative to either side of a plane. Again, comparable to the line equation, it can be proved that if $\mathbf{n}$ is normalized, $|ax + by + cz + d|$ returns the distance from the point to the plane.

Testing points versus planes using the general plane equation happens quite often. For example, to detect whether a point lies inside a convex polyhedron, you can do a plane test for every face of the polyhedron. Assuming the plane normals point away from the center of the polyhedron, if the point is on the negative side of all the planes then it lies inside. We may also use planes as culling devices that cut our world into half-spaces. If an object lies on one side of a plane, we consider it (say, for rendering purposes); otherwise, we ignore it. The distance property can be used to test whether a sphere is intersecting a plane. If the distance between the sphere's center and the plane is less than the sphere's radius, then the sphere is intersecting the plane.

Given three points in $\mathbb{R}^3$, P, Q, and R, we generate the generalized plane equation as follows. First we compute two vectors $\mathbf{u}$ and $\mathbf{v}$, where

$$\begin{aligned} \mathbf{u} &= Q - P \\ \mathbf{v} &= R - P \end{aligned}$$

Now we take the cross product of these two vectors to get the normal to the plane:

$$\mathbf{n} = \mathbf{u} \times \mathbf{v}$$

We usually normalize **n** at this point so that we can take advantage of the distance-measuring properties of the plane equation. This gives us our values a, b, and c. Taking P as the point on the plane, we compute d by

$$d = -(aP_x + bP_y + cP_z)$$

We can also use this to convert our parameterized form to the generalized form by starting with the cross product step.

Source Code Library IvMath Filename IvPlane

Since we'll be working in $\mathbb{R}^3$ most of the time and because of its useful properties, we'll be using the generalized plane equation as the basis for our class:

```
class IvPlane
{
public:
  IvPlane( float a, float b, float c, float d );

  IvVector3 mNormal;
  float    mOffset;
};
```

And while we'll be using this as our standard plane, from time to time we'll be making use of the parameterized form, so it's good to keep it in mind.

2.5.3 Coplanar Points

Four or more points are said to be *coplanar* if they all lie on a plane. Another way to think of this is that despite the number of points being greater than three, the affine space that they span is only two dimensional.

To determine whether four points P_0, P_1, P_2, and P_3 are coplanar, we create vectors $P_1 - P_0$, $P_2 - P_0$, and $P_3 - P_0$, and compute their triple scalar product. If the result is near zero, then they may be coplanar, if they're not collinear. To determine if they are collinear, take the cross products $(P_1 - P_0) \times (P_2 - P_0)$, and $(P_1 - P_0) \times (P_3 - P_0)$. If both results are near zero, then the points are collinear instead.

2.6 Polygons and Triangles

Source Code Library IvMath Filename IvTriangle

The current class of graphics processors wants their geometric data in primarily one form: points. However, having just a collection of points is not enough. We need to organize these points into smaller groups, for both rendering and computational purposes.

A *polygon* is made up of a set of *vertices* (which are represented by points) and *edges* (which are represented by line segments). The edges define how the vertices are connected together. A *convex polygon* is one where the set of points enclosed by the vertices and edges is a convex set; otherwise, it's a *concave polygon*.

The most commonly used polygons for storing geometric data are *triangles* (three vertices) and *quadrilaterals* (four vertices). While some rendering systems accept quadrilaterals (also referred to as just *quads*) as data, most want geometry grouped in triangles, so we'll follow that convention throughout the remainder of the book. One advantage triangles have over quadrilaterals is that three noncollinear vertices are guaranteed to be coplanar, so they can be used to define a single plane. If the three vertices of a triangle are collinear, then we have a *degenerate triangle*. Degenerate triangles can cause problems on some hardware and with some geometric algorithms, so it's good to cull them by checking for collinearity of the triangle vertices by using the technique described previously.

If the points are not collinear, then as we've stated, the three vertices P_0, P_1, and P_2 can be used to find the triangle's incident plane. If we set $\mathbf{u} = P_1 - P_0$ and $\mathbf{v} = P_2 - P_0$, then we can define this via the parameterized plane equation $P(s,t) = P_0 + s\mathbf{u} + t\mathbf{v}$. Alternately, we can compute the generalized plane equation by computing the cross product of $\mathbf{u}$ and $\mathbf{v}$, normalizing to get the normal $\hat{\mathbf{n}}$, and then computing d as described in Section 2.5.2.

It's often necessary to test whether a 3D point lying on the triangle plane is inside or outside of the triangle itself (Figure 2.38). We begin by computing three vectors $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{v}_2$, where

$$\mathbf{v}_0 = P_1 - P_0$$
$$\mathbf{v}_1 = P_2 - P_1$$
$$\mathbf{v}_2 = P_0 - P_2$$

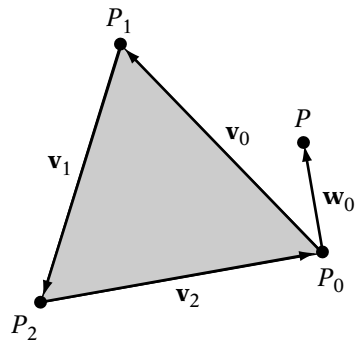


FIGURE 2.38 Point in triangle test.

We take the cross product of $\mathbf{v}_0$ and $\mathbf{v}_1$ to get a normal vector $\mathbf{n}$ to the triangle. We then compute three vectors from each vertex to the test point:

$$\begin{aligned}\mathbf{w}_0 &= P - P_0\\ \mathbf{w}_1 &= P - P_1\\ \mathbf{w}_2 &= P - P_2\end{aligned}$$

If the point lies inside the triangle, then the cross product of each $\mathbf{v}_i$ with each $\mathbf{w}_i$ will point in the same direction as $\mathbf{n}$, which we can test by using a dot product. If the result is negative, then we know they're pointing in opposite directions, and the point lies outside. For example, in Figure 2.38, the normal vector to the triangle, computed as $\mathbf{v}_0 \times \mathbf{v}_1$, points out of the page. But the cross product $\mathbf{v}_0 \times \mathbf{w}_0$ points into the page, so the point lies outside.

We can speed up this operation by projecting the point and triangle to one of the xy, xz, or yz planes and treating it as a 2D problem. To improve our accuracy, we'll choose the one that provides the maximum area for the projection of the triangle. If we look at the normal $\mathbf{n}$ for the triangle, one of the coordinate values (x, y, z) will have the maximum absolute value; that is, the normal is pointing generally along that axis. If we drop that coordinate and keep the other two, that will give us the maximum projected area. We can then throw out a number of zero terms and end up with a considerably faster test. This is equivalent to using the perpendicular dot product instead of the cross product. More detail on this technique can be found in Section 12.3.5, Triangles.

Another advantage that triangles have over quads is that (again, assuming the vertices aren't collinear) they are convex polygons. In particular, the convex combination of the three triangle vertices spans all the points that make up the triangle. Given a point P inside the triangle and on the triangle plane, it is possible to compute its particular barycentric coordinates (s, t), as used in the parameterized plane equation $P(s, t) = P_0 + s\mathbf{u} + t\mathbf{v}$. If we compute a vector $\mathbf{w} = P - P_0$, then we can rewrite the plane equation as

$$\begin{aligned}P &= P_0 + s\mathbf{u} + t\mathbf{v}\\ \mathbf{w} &= s\mathbf{u} + t\mathbf{v}\end{aligned}$$

If we take the cross product of $\mathbf{v}$ with $\mathbf{w}$, we get

$$\begin{aligned}\mathbf{v} \times \mathbf{w} &= \mathbf{v} \times (s\mathbf{u} + t\mathbf{v})\\ &= s(\mathbf{v} \times \mathbf{u}) + t(\mathbf{v} \times \mathbf{v})\\ &= s(\mathbf{v} \times \mathbf{u})\end{aligned}$$

Taking the length of both sides gives

$$\|\mathbf{v} \times \mathbf{w}\| = |s|\|\mathbf{v} \times \mathbf{u}\|$$

The quantity $\|\mathbf{v} \times \mathbf{u}\| = \|\mathbf{u} \times \mathbf{v}\|$. And since P is inside the triangle, we know that to meet the requirements of a convex combination $s \geq 0$, so

$$s = \frac{\|\mathbf{v} \times \mathbf{w}\|}{\|\mathbf{u} \times \mathbf{v}\|}$$

A similar construction finds that

$$t = \frac{\|\mathbf{u} \times \mathbf{w}\|}{\|\mathbf{u} \times \mathbf{v}\|}$$

Note that this is equivalent to computing the areas a and b of the two subtriangles shown in Figure 2.39 and dividing by the total area of the triangle c, so

$$s = \frac{b}{c}$$
$$t = \frac{a}{c}$$

where

$$a = \frac{1}{2}\|\mathbf{u} \times \mathbf{w}\|$$
$$b = \frac{1}{2}\|\mathbf{v} \times \mathbf{w}\|$$
$$c = \frac{1}{2}\|\mathbf{u} \times \mathbf{v}\|$$

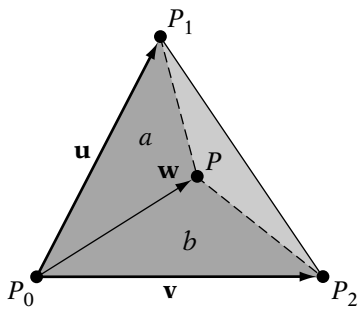


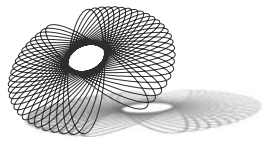
FIGURE 2.39 Computing barycentric coordinates for point in triangle.

These simple examples are only a taste of how we can use triangles in mathematical calculations. More details on the use and implementation of triangles can be found throughout the text, particularly in Chapters 7 and 12.

2.7 Chapter Summary

In this chapter, we have covered some basic geometric entities: vectors and points. We have discussed linear and affine spaces, the relationships between them, and how we can use affine combinations of vectors and points to define other entities like lines and planes. We've also shown how we can use our knowledge of affine spaces and vector properties to compute some simple tests on triangles. These skills will prove useful to us throughout the remainder of the text.

For those who are interested in reading further, Anton and Rorres [3] is a standard reference for many first courses in linear algebra. Other texts with slightly different approaches are Axler [4] and Friedberg et al. [39]. Information on points and affine spaces can be found in Schneider and Eberly [100], as well as in deRose [23].

Chapter 3

Matrices and Linear Transformations

3.1 Introduction

In the previous chapter we discussed vectors and points and some simple operations we can apply to them. Now we'll begin to expand our discussion to cover specific functions that we can apply to vectors and points; functions known as transformations. In this chapter we'll discuss a class of transformations that we can apply to vectors called *linear transformations*. These encompass nearly all of the common operations we might want to perform on vectors and points, so understanding what they are and how to apply them is important. We'll define these functions and how they are distinguished from other, more general transformations.

Properties of linear transformations allow us to use a structure called a *matrix* as a compact representation for transforming vectors. A matrix is a simple two-dimensional (2D) array of values, but within it lies all the power of a linear transformation. Through simple operations we can use the matrix to apply linear transformations to vectors. We can also combine two transformation matrices to create a new one that has the same effect of the first two. Using matrices effectively lies at the heart of the pipeline for manipulating virtual objects and rendering them on the screen.

Matrices have other applications as well. Examining the structure of a matrix can tell us something about the transformation it represents, for example, whether it can be reversed, what that reverse transformation might be, or whether it distorts the data that it is given. Matrices also can be used

to solve systems of linear equations, which is useful to know for certain algorithms in graphics and physical simulation. For all of these reasons, matrices are primary data structures in graphics application programmer interfaces (APIs).

3.2 Matrices

3.2.1 Introduction to Matrices

A matrix is a rectangular, 2D array of values. Throughout this book, most of the values we use will be real numbers, but they could be complex numbers or even vectors. Each individual value in a matrix is called an *element*. Examples of matrices are

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 35 & -15 \\ 2 & 52 & 1 \end{bmatrix} \quad \mathbf{C} = \begin{bmatrix} 2 & -1 \\ 0 & 2 \\ 6 & 3 \end{bmatrix}$$

A matrix is described as having m rows by n columns, or being an $m \times n$ matrix. A row is a horizontal group of elements from left to right, while a column is a vertical, top-to-bottom group. Matrix **A** in our example has 3 rows and 3 columns and is a 3×3 matrix, whereas matrix **C** is a 3×2 matrix. Rows are numbered 0 to $m-1$,[1] while columns are numbered 0 to $n-1$. An individual element of a matrix **A** is referenced as either $(\mathbf{A})_{i,j}$ or just $a_{i,j}$, where i is the row number and j is the column. Looking at matrix **B**, element $b_{1,0}$ contains the value 2 and element $b_{0,1}$ equals 35.

If an individual matrix has an equal number of rows and columns, that is if $m = n$, then it is called a *square matrix*. In our example, matrix **A** is square, whereas matrices **B** and **C** are not.

If all elements of a matrix are zero, then it is called a *zero matrix*. We will represent a matrix of this type as **0** and assume a matrix of the appropriate size for the operation we are performing.

If two matrices have an equal number of rows and columns, then they are said to be the same *size*. If they are the same size and their corresponding

1. As a reminder, mathematical convention starts with 1, but we're using 0 to be compatible with C++.

elements have the same values, then they are *equal*. Below, the two matrices are the same size, but they are not equal.

$$\begin{bmatrix} 0 & 1 \\ 3 & 2 \\ 0 & -3 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 \\ 2 & -3 \\ 1 & 3 \end{bmatrix}$$

The set of elements where the row and column numbers are the same (e.g., row 1, column 1) is called the *main diagonal*. In the next example the main diagonal is in gray.

$$\mathbf{U} = \begin{bmatrix} 3 & -5 & 0 & 1 \\ 0 & 2 & 6 & 0 \\ 0 & 0 & 1 & -8 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The *trace* of a matrix is the sum of the main diagonal elements. In this case the trace is $3 + 2 + 1 + 1 = 7$.

In matrix **U**, all elements below the diagonal are equal to 0. This is known as an *upper triangular matrix*. Note that elements above the diagonal don't necessarily have to be nonzero in order for the matrix to be upper triangular, nor does the matrix have to be square.

If elements above the diagonal are 0, then we have a *lower triangular matrix*:

$$\mathbf{L} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ -6 & 1 & 0 & 1 \end{bmatrix}$$

Finally, if a square matrix has nondiagonal elements of zero, we call the matrix a *diagonal matrix*:

$$\mathbf{D} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It follows that any diagonal matrix is both an upper triangular and lower triangular matrix.

3.2.2 Simple Operations

Matrix Addition and Scalar Multiplication

We can add and scale matrices just as we can vectors. Adding two matrices together:

$$\mathbf{S} = \mathbf{A} + \mathbf{B}$$

is done componentwise like vectors, thus,

$$s_{i,j} = a_{i,j} + b_{i,j}$$

Clearly, in order for this to work, **A**, **B**, and **S** must all be the same size (also known as *conformable for addition*). Subtraction works similarly but as with real numbers and vectors is not commutative.

To scale a matrix,

$$\mathbf{P} = s\mathbf{A}$$

each element is multiplied by the scalar, again like vectors:

$$p_{i,j} = s \cdot a_{i,j}$$

Matrix addition and scalar multiplication have their algebraic rules, which should seem quite familiar at this point:

1. $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$.
2. $\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$.
3. $\mathbf{A} + \mathbf{0} = \mathbf{A}$.
4. $\mathbf{A} + (-\mathbf{A}) = \mathbf{0}$.
5. $a(\mathbf{A} + \mathbf{B}) = a\mathbf{A} + a\mathbf{B}$.
6. $a(b\mathbf{A}) = (ab)\mathbf{A}$.
7. $(a + b)\mathbf{A} = a\mathbf{A} + b\mathbf{A}$.
8. $1\mathbf{A} = \mathbf{A}$.

As we can see, these rules match the requirements for a vector space, and so the set of matrices of a given size is also a vector space.

Transpose

The *transpose* of a matrix $\mathbf{A}$ (represented by $\mathbf{A}^T$) interchanges the rows and columns of $\mathbf{A}$. It does this by exchanging elements across the matrix's main diagonal, so $(\mathbf{A}^T)_{i,j} = (\mathbf{A})_{j,i}$. An example of this is

$$\begin{bmatrix} 2 & -1 \\ 0 & 2 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 6 \\ -1 & 2 & 3 \end{bmatrix}$$

As we can see, the matrix does not have to be square, so an $m \times n$ matrix becomes an $n \times m$ matrix. Also, the main diagonal doesn't change, or is invariant, since $(\mathbf{A}^T)_{i,i} = (\mathbf{A})_{i,i}$.

A matrix where $(\mathbf{A})_{i,j} = (\mathbf{A})_{j,i}$ (i.e., cross-diagonal entries are equal) is called a *symmetric matrix*. All diagonal matrices are symmetric. Another example of a symmetric matrix is

$$\begin{bmatrix} 3 & 1 & 2 & 3 \\ 1 & 2 & -5 & 0 \\ 2 & -5 & 1 & -9 \\ 3 & 0 & -9 & 1 \end{bmatrix}$$

The transpose of a symmetric matrix is the matrix again, since in this case $(\mathbf{A}^T)_{j,i} = (\mathbf{A})_{i,j} = (\mathbf{A})_{j,i}$.

A matrix where $(\mathbf{A})_{i,j} = -(\mathbf{A})_{j,i}$ (i.e., cross-diagonal entries are negated and the diagonal is 0) is called a *skew symmetric matrix*. An example of a skew symmetric matrix is

$$\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & -5 \\ -2 & 5 & 0 \end{bmatrix}$$

The transpose of a skew symmetric matrix is the negation of the original matrix, since in this case $(\mathbf{A}^T)_{j,i} = (\mathbf{A})_{i,j} = -(\mathbf{A})_{j,i}$.

Some useful algebraic rules involving the transpose are

1. $(\mathbf{A}^T)^T = \mathbf{A}$
2. $(a\mathbf{A}^T) = a\mathbf{A}^T$
3. $(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$

where a is a scalar and $\mathbf{A}$ and $\mathbf{B}$ are conformable for addition.

3.2.3 Vector Representation

If a matrix has only one row or one column, then we have a row or column matrix, respectively:

$$\begin{bmatrix} 0.5 & 0.25 & 1 & -1 \end{bmatrix} \quad \begin{bmatrix} 5 \\ -3 \\ 6.9 \end{bmatrix}$$

These are often used to represent vectors. There is no particular standard as to which one to use. For example, the OpenGL specification and its documentation uses columns, whereas DirectX, by comparison, uses rows. In this text we will assume that vectors are represented as column matrices (also known as column vectors). First of all, most math texts use column vectors and we wish to remain compatible. In addition, the classical presentation of quaternions (another means for performing some linear transformations) uses a concatenation order consistent with the use of column matrices for vectors.

The choice to represent vectors as column matrices does have some effect on how we construct and multiply our matrices, which we will discuss in more detail in the following parts. In the cases where we do wish to indicate that a vector is represented as a row matrix, we'll display it with a transpose applied, like $\mathbf{b}^T$.

3.2.4 Block Matrices

A matrix also can be represented by submatrices, rather than by individual elements. This is also known as a *block matrix*. For example, the matrix

$$\begin{bmatrix} 2 & 3 & 0 \\ -3 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

also can be represented as

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ -3 & 2 \end{bmatrix}$$

and

$$\mathbf{0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We will sometimes use this to represent a matrix as a set of row or column matrices. For example, if we have a matrix $\mathbf{A}$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix}$$

we can represent its rows as three vectors

$$\begin{aligned} \mathbf{a}_0^T &= \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \end{bmatrix} \\ \mathbf{a}_1^T &= \begin{bmatrix} a_{1,0} & a_{1,1} & a_{1,2} \end{bmatrix} \\ \mathbf{a}_2^T &= \begin{bmatrix} a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \end{aligned}$$

and represent $\mathbf{A}$ as

$$\begin{bmatrix} \mathbf{a}_0^T \\ \mathbf{a}_1^T \\ \mathbf{a}_2^T \end{bmatrix}$$

Similarly, we can represent a matrix $\mathbf{B}$ with its columns as three vectors

$$\begin{aligned} \mathbf{b}_0 &= \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \end{bmatrix} \\ \mathbf{b}_1 &= \begin{bmatrix} b_{0,1} \\ b_{1,1} \\ b_{2,1} \end{bmatrix} \\ \mathbf{b}_2 &= \begin{bmatrix} b_{0,2} \\ b_{1,2} \\ b_{2,2} \end{bmatrix} \end{aligned}$$

and subsequently $\mathbf{B}$ as

$$\begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 \end{bmatrix}$$

As mentioned earlier, the transpose notation tells us whether we're using row or column vectors.

3.2.5 Matrix Product

The primary operation we will apply to matrices is multiplication, also known as the *matrix product*. The product is important to us because it allows us to do two essential things. First, multiplying a matrix by a compatible vector will *transform* the vector. Second, multiplying matrices together will create a single matrix that performs their combined transformations. We'll discuss exactly what is occurring when we discuss linear transformations below, but for now we 'll just define how to perform matrix multiplication.

As with real numbers, the product **C** of two matrices **A** and **B** is represented as

$$\mathbf{C} = \mathbf{AB}$$

Computing the matrix product is not as simple as multiplying real numbers but is not that bad if you understand the process. To calculate a given element $c_{i,j}$ in the product, we take the dot product of row i from **A** with column j from **B**. We can express this symbolically as

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

As an example, we'll look at computing the first element of a 3×3 matrix:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} b_{0,0} & \cdots & \cdots \\ b_{1,0} & \cdots & \cdots \\ b_{2,0} & \cdots & \cdots \end{bmatrix} = \begin{bmatrix} c_{0,0} & \cdots & \cdots \\ \vdots & \ddots & \vdots \\ \vdots & \cdots & \ddots \end{bmatrix}$$

To compute the value of $c_{0,0}$, we take the dot product of row 1 from **A** and column 1 from **B**:

$$c_{0,0} = a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + a_{0,2}b_{2,0}$$

Expanding this for a 2×2 matrix:

$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} = \begin{bmatrix} a_{0,0}b_{0,0} + a_{0,1}b_{1,0} & a_{0,0}b_{0,1} + a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0} + a_{1,1}b_{1,0} & a_{1,0}b_{0,1} + a_{1,1}b_{1,1} \end{bmatrix}$$

If we represent **A** as a collection of rows and **B** as a collection of columns, then

$$\begin{bmatrix} \mathbf{a}_0^T \\ \mathbf{a}_1^T \end{bmatrix} \begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{a}_0 \cdot \mathbf{b}_0 & \mathbf{a}_0 \cdot \mathbf{b}_1 \\ \mathbf{a}_1 \cdot \mathbf{b}_0 & \mathbf{a}_1 \cdot \mathbf{b}_1 \end{bmatrix}$$

We can also multiply by using block matrices:

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{bmatrix} = \begin{bmatrix} \mathbf{AE} + \mathbf{BG} & \mathbf{AF} + \mathbf{BH} \\ \mathbf{CE} + \mathbf{DG} & \mathbf{CF} + \mathbf{DH} \end{bmatrix}$$

Note that this is only allowable if the submatrices are conformable for addition and multiplication.

There is a restriction on which matrices can be multiplied together; in order to perform a dot product the two vectors have to have the same length. So, to multiply together two matrices, the number of columns in the first (i.e., the width of each row) has to be the same as the number of rows in the second (i.e., the height of each column). Because of this restriction, only square matrices can be multiplied by themselves.

As previously indicated, matrices can be used to transform vectors. We do this by multiplying the matrix by a column matrix representing the vector we wish to transform, or:

$$\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

We can represent this in matrix–vector notation as just

$$\mathbf{b} = \mathbf{A}\mathbf{x}$$

Note that in this case the number of columns in the matrix must match the number of elements in the vector.

Column vectors aren't the only possibility. We can also premultiply by a vector by treating it as a row matrix:

$$\begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & \cdots & x_{m-1} \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

or

$$\mathbf{c}^T = \mathbf{x}^T\mathbf{A}$$

And now note that in this case the number of rows in the matrix must match the number of elements in the vector.

In general, matrix multiplication is not commutative. As an example, if we multiply a row matrix by a column matrix, we perform a dot product:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = 1 \cdot 3 + 2 \cdot 4 = 11$$

Because of this, you may often see a dot product represented as

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$$

If we multiply them in the opposite order, we get a square matrix:

$$\begin{bmatrix} 3 \\ 4 \end{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix}$$

Even multiplication of square matrices is not necessarily commutative:

$$\begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 9 & 6 \\ 12 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 7 & 14 \end{bmatrix}$$

Aside from the size restriction and not being commutative, the algebraic rules for matrix multiplication are very similar to those for real numbers:

1. $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$
2. $a(\mathbf{BC}) = (a\mathbf{B})\mathbf{C}$
3. $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$
4. $(\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$
5. $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$

where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are matrices conformable for multiplication and a is a scalar. Note that matrix multiplication is still associative (rules 1 and 2) and distributive (rules 3 and 4).

3.2.6 Identity Matrix

We know that when we multiply a scalar or vector by 1, the result is the scalar or vector again:

$$1 \cdot x = x$$

Similarly, in matrix multiplication there is a special matrix known as the *identity matrix*, represented by the letter **I**. Thus,

$$\mathbf{A} \cdot \mathbf{I} = \mathbf{I} \cdot \mathbf{A} = \mathbf{A}$$

A particular identity matrix is a diagonal square matrix, where the diagonal is all 1s:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

If a particular $n \times n$ identity matrix is needed, it is sometimes referred to as $\mathbf{I}_n$. Take as an example $\mathbf{I}_3$:

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rather than referring to it in this way, we'll just use the term **I** to represent a general identity matrix and assume it is the correct size in order to allow an operation to proceed.

3.2.7 Performing Vector Operations with Matrices

Recall that if we multiply a row vector by a column vector, it performs a dot product:

$$\mathbf{w}^T \mathbf{v} = w_x v_x + w_y v_y + w_z v_z = \mathbf{v} \cdot \mathbf{w}$$

And multiplying them in the opposite order produces a square matrix:

$$\mathbf{T} = \mathbf{v}\,\mathbf{w}^T = \begin{bmatrix} v_x w_x & v_x w_y & v_x w_z \\ v_y w_x & v_y w_y & v_y w_z \\ v_z w_x & v_z w_y & v_z w_z \end{bmatrix}$$

This square matrix **T** is known as the *tensor product* $\mathbf{v} \otimes \mathbf{w}$. We can use it to rewrite vector expressions of the form $(\mathbf{u} \cdot \mathbf{v})\,\mathbf{w}$ as

$$(\mathbf{u} \cdot \mathbf{v})\,\mathbf{w} = (\mathbf{w} \otimes \mathbf{v})\,\mathbf{u}$$

In particular, we can rewrite a projection by a unit vector as

$$(\mathbf{u} \cdot \hat{\mathbf{v}})\hat{\mathbf{v}} = (\hat{\mathbf{v}} \otimes \hat{\mathbf{v}})\mathbf{u}$$

This will prove useful to us in the next chapter.

We can also perform our other vector product, the cross product, through a matrix multiplication. If we have two vectors $\mathbf{v}$ and $\mathbf{w}$ and we want to compute $\mathbf{v} \times \mathbf{w}$, we can replace $\mathbf{v}$ with a particular skew symmetric matrix, represented as $\tilde{\mathbf{v}}$:

$$\tilde{\mathbf{v}} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

Multiplying by $\mathbf{w}$ gives

$$\begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} = \begin{bmatrix} v_y w_z - w_y v_z \\ v_z w_x - w_z v_x \\ v_x w_y - w_x v_y \end{bmatrix}$$

which is the formula for the cross product. This will also prove useful to us in subsequent chapters.

3.2.8 Implementation

Source Code
Library
IvMath
Filename
IvMatrix33
IvMatrix44

One might expect that the most natural data format for, say, a 3×3 matrix would be

```
class IvMatrix33
{
   float mData[3][3];
};
```

However, the memory layout of such a matrix is not ideal for our purposes. In C or C++, 2D arrays are stored in what is called *row major order*, meaning that the matrix is stored in memory in a row-by-row order. If we use a one-dimensional (1D) array as our member variable instead:

```
class IvMatrix33
{
   float mV[9];
};
```

the index order for a 3×3 matrix is

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

The indexing operator for a row major matrix (we have to use `operator()` because `operator[]` only works for a single index) is

```
float&
IvMatrix33::operator()(unsigned int row, unsigned int col)
{
  return mV[col + 3*row];
}
```

Why won't this work? Well, in Direct3D matrices are expected to be used with row vectors. And even in OpenGL, despite the fact that the documentation is written using column vectors, the internal representation premultiplies the vectors; that is, it expects row vectors as well. Accordingly, since we're using column vectors, we will need to transpose our matrices before we pass them in as arguments to the graphics API. Doing this for every single matrix takes time and is a bit of a nuisance to remember. Missing that one transpose can make debugging your algorithm a longer process than it needs to be.

The solution is to pretranspose the matrix in the storage representation. This is a format known as *column major order* and stores a matrix column by column instead of row by row. Writing out our indices in column major order gives us

$$\begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$$

Notice that the indices are the transpose of row major order. The indexing operator becomes

```
float&
IvMatrix33::operator()(unsigned int row, unsigned int col)
{
  return mV[row + 3*col];
}
```

Alternatively, if we want to use 2D arrays:

```
float&
IvMatrix33::operator()(unsigned int row, unsigned int col)
{
  return mV[col][row];
}
```

Using column major format and column vectors, matrix–vector multiplication becomes

```
IvVector3
IvMatrix33::operator*( const IvVector3& vector ) const
{
    IvVector3 result;

    result.x = mV[0]*vector.x + mV[3]*vector.y + mV[6]*vector.z;
    result.y = mV[1]*vector.x + mV[4]*vector.y + mV[7]*vector.z;
    result.z = mV[2]*vector.x + mV[5]*vector.y + mV[8]*vector.z;

    return result;
}
```

and matrix–matrix multiplication is

```
IvMatrix33
IvMatrix33::operator*( const IvMatrix33& other ) const
{
    IvMatrix33 result;

    result.mV[0] = mV[0]*other.mV[0] + mV[3]*other.mV[1] + mV[6]*other.mV[2];
    result.mV[1] = mV[1]*other.mV[0] + mV[4]*other.mV[1] + mV[7]*other.mV[2];
    result.mV[2] = mV[2]*other.mV[0] + mV[5]*other.mV[1] + mV[8]*other.mV[2];

    result.mV[3] = mV[0]*other.mV[3] + mV[3]*other.mV[4] + mV[6]*other.mV[5];
    result.mV[4] = mV[1]*other.mV[3] + mV[4]*other.mV[4] + mV[7]*other.mV[5];
    result.mV[5] = mV[2]*other.mV[3] + mV[5]*other.mV[4] + mV[8]*other.mV[5];

    result.mV[6] = mV[0]*other.mV[6] + mV[3]*other.mV[7] + mV[6]*other.mV[8];
    result.mV[7] = mV[1]*other.mV[6] + mV[4]*other.mV[7] + mV[7]*other.mV[8];
    result.mV[8] = mV[2]*other.mV[6] + mV[5]*other.mV[7] + mV[8]*other.mV[8];

    return result;
}
```

Matrix addition is just

```
IvMatrix33
IvMatrix33::operator+( const IvMatrix33& other ) const
{
    IvMatrix33 result;
    for (int i = 0; i < 9; ++i)
    {
        result.mV[i] = mV[i]+other.mV[i];
    }
    return result;
}
```

Scalar multiplication of matrices is similar.

It is common practice to refer to a matrix intended to be used with row vectors (i.e., its transformed basis vectors are stored as rows) as row major order and, similarly, to a matrix intended to be used with column vectors as column major order. This is incorrect terminology. Row and column major order refer only to the storage format; namely, where an element $a_{i,j}$ will lie in the 1D representation of the matrix. Whether your matrix library intends for vectors to be pre- or postmultiplied should be independent of the underlying storage.

3.3 Linear Transformations

Now that we've discussed the structure and basic functionality of matrices, we can discuss their purpose as an engine for performing linear transformations. Linear transformations are a very useful and important concept in linear algebra. As one of a class of functions known as transformations, they map vector spaces to vector spaces. This allows us to apply complex functions to, or *transform*, vectors. Linear transformations perform this mapping while also having the additional property of preserving linear combinations. We will see how this permits us to describe a linear transformation in terms of how it affects the basis vectors of a vector space. Later sections will show how this in turn allows us to use matrices to represent linear transformations.

3.3.1 Definitions

Before we can begin to discuss transformations and linear transformations in particular, we need to define a few terms. A *relation* maps a set X of values (known as the *domain*) to another set Y of values (known as the *range*).

A *function* is a relation where every value in the first set maps to one and only one value in the second set, for example, $f(x) = \sin x$. An example of a relation that is not a function is $\pm\sqrt{x}$, because there are two possible results for a positive value of x, either positive or negative.

A function whose domain is an n-dimensional space and whose range is an m-dimensional space is known as a *transformation*. A transformation that maps from $\mathbb{R}^n$ to $\mathbb{R}^m$ is expressed as $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^m$. If the domain and the range of a transformation are equal (i.e., $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^n$), then the transformation is sometimes called an *operator*.

An example of a transformation is the function

$$f(x, y) = x^2 + 2y$$

which maps from $\mathbb{R}^2$ to $\mathbb{R}$. Another example is

$$f(x, y, z) = x^2 + 2y + \sqrt{z}$$

which maps from $\mathbb{R}^3$ to $\mathbb{R}$.

We can also map to a multidimensional space. For example, we could define a transformation from $\mathbb{R}^2$ to $\mathbb{R}^2$ as follows:

$$\mathcal{T}(a, b) = (f(a, b), g(a, b)) \tag{3.1}$$

A *linear transformation* $\mathcal{T}$ is a mapping between two vector spaces V and W, where for all $\mathbf{v}$ in V and for all scalars a:

1. $\mathcal{T}(\mathbf{v}_0 + \mathbf{v}_1) = \mathcal{T}(\mathbf{v}_0) + \mathcal{T}(\mathbf{v}_1)$ for all $\mathbf{v}_0$, $\mathbf{v}_1$ in V.
2. $\mathcal{T}(a\mathbf{v}) = aT(\mathbf{v})$ for all $\mathbf{v}$ in V.

To determine whether a transformation is linear, it is sufficient to show that

$$\mathcal{T}(a\mathbf{x} + \mathbf{y}) = a\mathcal{T}(\mathbf{x}) + \mathcal{T}(\mathbf{y})$$

An example of a linear transformation is $\mathcal{T}(\mathbf{x}) = k\mathbf{x}$, where k is any fixed scalar. We can show this by

$$\begin{aligned}\mathcal{T}(a\mathbf{x} + \mathbf{y}) &= k(a\mathbf{x} + \mathbf{y}) \\ &= ak\mathbf{x} + k\mathbf{y} \\ &= a\mathcal{T}(\mathbf{x}) + \mathcal{T}(\mathbf{y})\end{aligned}$$

On the other hand, the function $g(x) = x^2$ is not linear because, for $a = 2$, $x = 1$, and $y = 1$:

$$\begin{aligned} g(2(1) + 1) &= (2(1) + 1)^2 \\ &= 3^2 = 9 \\ &\neq 2(g(1)) + g(1) \\ &= 2(1^2) + 1^2 = 3 \end{aligned}$$

As we might expect, the only operations possible in a linear function are multiplication by a constant and addition.

3.3.2 Null Space and Range

We define the *null space* (or *kernel*) $N(\mathcal{T})$ of a linear transformation $\mathcal{T} : V \to W$ as the set of all vectors in V that map to $\mathbf{0}$, or

$$N(\mathcal{T}) = \{\mathbf{x} \mid \mathcal{T}(\mathbf{x}) = \mathbf{0}\}$$

The dimension of $N(\mathcal{T})$ is called the *nullity* of the transformation.

We formally define the *range* $R(\mathcal{T})$ of a linear transformation $\mathcal{T} : V \to W$ as the set of all vectors in W that are mapped to by at least one vector in V, or

$$R(\mathcal{T}) = \{\mathcal{T}(\mathbf{x}) | \mathbf{x} \in V\}$$

The dimension of $R(\mathcal{T})$ is called the *rank* of the transformation.

The null space and range have two important properties. First of all, they are both vector spaces, and in fact the null space is a subspace of V and the range is a subspace of W. Second,

$$nullity(\mathcal{T}) + rank(\mathcal{T}) = dim(V)$$

To get a better sense of this, let's look at an example. Suppose we have the linear transformation

$$\mathcal{T}(a, b) = (a + b, 0)$$

The resulting range space is of the form $(x, 0)$, so it can be spanned by the vector $(1, 0)$ and has dimension 1. The transformation will produce the vector $(0, 0)$ only when $a = -b$. So the null space has a basis of $(1, -1)$ and is also one dimensional. As we expect, they add up to 2, the dimension of our original vector space (Figure 3.1).

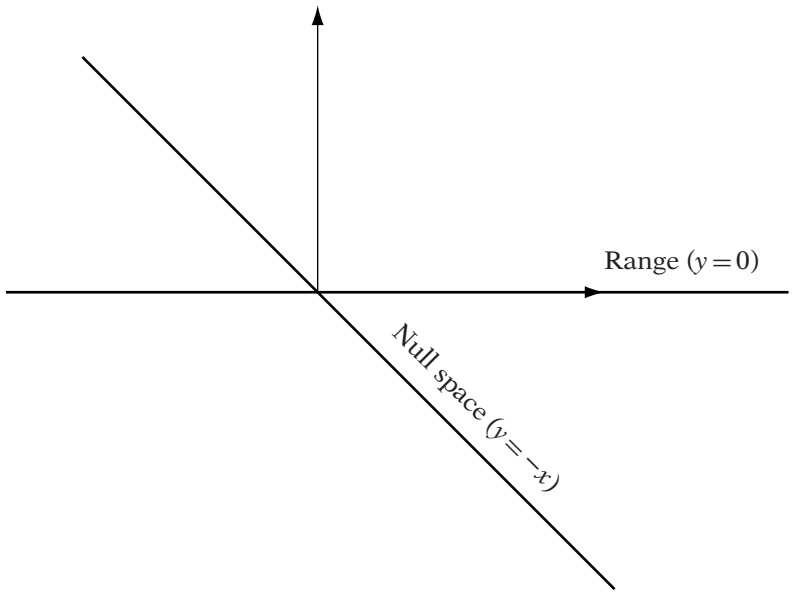


FIGURE 3.1 Range and null space for transformation $\mathtt{T}(a, b) = (a + b, 0)$.

3.3.3 Linear Transformations and Basis Vectors

Using standard function notation to represent linear transformations (as in equation 3.1) is not the most convenient or compact format, particularly for transformations between higher-dimensional vector spaces. Let's examine the properties of vectors as they undergo a linear transformation and see how that can lead us to a better representation.

Recall that we can represent any vector $\mathbf{x}$ in an n-dimensional vector space V as

$$\mathbf{x} = x_0\mathbf{v}_0 + x_1\mathbf{v}_1 + \cdots + x_{n-1}\mathbf{v}_{n-1}$$

where $\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}\}$ is a basis for V.

Now suppose we have a linear transformation $\mathcal{T} : V \to W$ that maps from V to an m-dimensional vector space W. If we apply our transformation to our arbitrary vector $\mathbf{x}$, then we have

$$\begin{aligned}\mathcal{T}(\mathbf{x}) &= \mathcal{T}(x_0\mathbf{v}_0 + x_1\mathbf{v}_1 + \cdots + x_{n-1}\mathbf{v}_{n-1}) \\ &= x_0\mathcal{T}(\mathbf{v}_0) + x_1\mathcal{T}(\mathbf{v}_1) + \cdots + x_{n-1}\mathcal{T}(\mathbf{v}_{n-1}) \qquad (3.2)\end{aligned}$$

So, if we know how our linear transformation affects our basis for V, then we can calculate the effect of the linear transformation for any arbitrary vector in V.

There is still an open question: What are the components of each $\mathcal{T}(\mathbf{v}_j)$ equal to? For a member $\mathbf{v}_j$ of V's basis, we can represent $\mathcal{T}(\mathbf{v}_j)$ in terms of the basis $\{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{m-1}\}$ for W, again as a linear combination:

$$\mathcal{T}(\mathbf{v}_j) = a_{0,j}\mathbf{w}_0 + a_{1,j}\mathbf{w}_1 + \cdots + a_{m-1,j}\mathbf{w}_{m-1}$$

If $\{\mathbf{w}_0, \ldots, \mathbf{w}_{m-1}\}$ is the standard basis for W, this simplifies to

$$\mathcal{T}(\mathbf{v}_j) = (a_{0,j}, a_{1,j}, \ldots, a_{m-1,j}) \tag{3.3}$$

Combining equations 3.2 and 3.3 gives us

$$\begin{aligned}\mathcal{T}(\mathbf{x}) &= x_0(a_{0,0}, a_{1,0}, \ldots, a_{m-1,0}) \\ &= +x_1(a_{0,1}, a_{1,1}, \ldots, a_{m-1,1}) \\ &= \cdots \\ &= +x_{n-1}(a_{0,n-1}, a_{1,n-1}, \ldots, a_{m-1,n-1})\end{aligned} \tag{3.4}$$

If we set $\mathbf{b} = \mathcal{T}(\mathbf{x})$, then for a given component of $\mathbf{b}$

$$b_i = a_{i,0}x_0 + a_{i,1}x_1 + \cdots + a_{i,n-1}x_{n-1} \tag{3.5}$$

Knowing this, we can precalculate and store the n transformed basis vectors $(a_{0,j}, a_{1,j}, \ldots, a_{m-1,j})$ and use this formula at any time to transform a general vector $\mathbf{x}$.

Let's look at an example taking a transformation from $\mathbb{R}^2$ to $\mathbb{R}^2$, using the standard basis for both vector spaces:

$$\mathcal{T}(a, b) = (a + b, b)$$

If we look at how this affects our standard basis for $\mathbb{R}^2$, we get

$$\begin{aligned}\mathcal{T}(1, 0) &= (1 + 0, 0) = (1, 0) \\ \mathcal{T}(0, 1) &= (0 + 1, 1) = (1, 1)\end{aligned}$$

Transforming an arbitrary vector in $\mathbb{R}^2$, say $(2, 3)$, we get

$$\begin{aligned}\mathcal{T}(2, 3) &= 2\mathcal{T}(1, 0) + 3\mathcal{T}(0, 1) \\ &= 2(1, 0) + 3(1, 1) \\ &= (5, 3)\end{aligned}$$

which is what we expect.

It should be made clear that applying a linear transformation to a basis does not produce the basis for the new vector space. It only shows where the basis vectors end up in the new vector space—in our case in terms of the standard basis. In fact, a transformed basis may be no longer linearly independent. Take as another example

$$\mathcal{T}(a, b) = (a + b, 0)$$

Applying this to our standard basis for $\mathbb{R}^2$, we get

$$\mathcal{T}(1, 0) = (1 + 0, 0) = (1, 0)$$

$$\mathcal{T}(0, 1) = (0 + 1, 0) = (1, 0)$$

The two resulting vectors are clearly linearly dependent.

These two examples illustrate one useful property. If the rank of a linear transformation $\mathcal{T}$ equals the number of elements in a transformed basis β, then we can say that β is linearly independent. In fact, the rank is equal to the number of linearly independent elements in β, and those linearly independent elements will span the range of $\mathcal{T}$.

3.3.4 Matrices and Linear Transformations

Knowing that we can represent a linear transformation in terms of how the basis vectors are transformed is a very powerful tool. As we will now see, it is precisely this property of linear transformations that allows us to represent them concisely by using a matrix.

Let's look again at a matrix–vector multiplication with our terms expanded:

$$\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Note that $\mathbf{x}$ has n components and the resulting vector $\mathbf{b}$ has m. In order for the multiplication to proceed, matrix $\mathbf{A}$ must be $m \times n$. This represents a transformation from an n-dimensional space V to an m-dimensional space W.

To see how this operation performs a linear transformation, we'll use the fact that we only need to know where the basis of a vector space V is mapped to. Suppose that we know that our standard basis $\{\mathbf{e}_0, \mathbf{e}_1, \ldots, \mathbf{e}_{n-1}\}$ is transformed to $\{\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{n-1}\}$ in W, again using the standard basis. We

will store, in order, each of these transformed basis vectors as the columns of **A**, or

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \cdots & \mathbf{a}_{n-1} \end{bmatrix}$$

Using our matrix multiplication definition to compute the product of **A** and a vector **x** in V, we see that the result for element i in **b** is

$$b_i = a_{i,0}x_0 + a_{i,1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

This is exactly the same as equation 3.5. So, by setting up our matrix with the transformed basis vectors in each column, we can use matrix multiplication to perform linear transformations.

This provides an explanation for the properties of the identity matrix: It maps the basis vectors of the domain to the same vectors in the range. Or to put it another way: It performs a linear transformation that has no effect on the source vector, also known as the identity transformation.

Recall that we can also premultiply by a vector by treating it as a row matrix:

$$\begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & \cdots & x_{m-1} \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

In this case, the rows of **A** are acting as our transformed basis vectors, and the number of components in $\mathbf{x}^T$ must match the number of rows in our matrix.

At this point we can define some additional properties for matrices. The *column space* of a matrix is the vector space spanned by the matrix's column vectors and is the range of the linear transformation performed by post-multiplying by a column vector. Correspondingly, the *row space* is the vector space spanned by the row vectors of the matrix and, as we'd expect, is the range of the linear transformation performed by premultiplying by a row vector. As it happens, the dimensions of the row space and column space are equal and that value is called the *rank* of the matrix. The matrix rank is equal to the rank of the associated linear transformation.

The column space and row space are not necessarily the same vector space. As an example, take the matrix

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

When postmultiplied by a column vector, it maps a vector (x, y, z) in $\mathbb{R}^3$ to a vector $(y, z, 0)$ on the xy plane. Premultiplying by a row vector, on the other hand, maps (x, y, z) to $(0, x, y)$ on the yz plane. They have the same dimension, and hence the same rank, but they are not the same vector space.

This makes a certain amount of sense. When we multiply by a row vector, we use the row vectors of the matrix as our transformed basis instead of the column vectors. To achieve the same result as the column vector multiplication, we need to change our matrix's column vectors to row vectors by taking the transpose:

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} y & z & 0 \end{bmatrix}$$

We can now see the purpose of the transpose: It exchanges a matrix's row space with its column space.

Like a linear transformation, a matrix also has a null space, which is all vectors $\mathbf{x}$ in V such that

$$\mathbf{A}\mathbf{x} = \mathbf{0}$$

In the preceding example, the null space N is all vectors with zero y and z components. As with linear transformations, $\dim(N) + \text{rank}\,(\mathbf{A}) = \dim(V)$.

3.3.5 Combining Linear Transformations

Suppose we have two transformations, $\mathcal{S} : U \to V$ and $\mathcal{T} : V \to W$, and we want to perform one after the other; namely, for a vector $\mathbf{x}$, we want the result $\mathcal{T}(\mathcal{S}(\mathbf{x}))$. If we know that we are going to transform a large collection of vectors by $\mathcal{S}$ and the resulting vectors by $\mathcal{T}$, it will be more efficient to find a single transformation that generates the same result so that we only have to transform the vectors once. This is known as the *composition* of $\mathcal{S}$ and $\mathcal{T}$ and is written as

$$(\mathcal{T} \circ \mathcal{S})(\mathbf{x}) = \mathcal{T}(\mathcal{S}(\mathbf{x}))$$

Composition (or alternatively, *concatenation*) of transformations is done via generalized matrix multiplication.

Suppose that matrix $\mathbf{A}$ is the corresponding transformation matrix for $\mathcal{S}$ and $\mathbf{B}$ is the corresponding matrix for $\mathcal{T}$. Recall that in order to set up $\mathbf{A}$ for vector transformation, we pretransform the standard basis vectors by $\mathcal{S}$ and store them as the columns of $\mathbf{A}$. Now we need to transform those vectors again, this time by $\mathcal{T}$. We could either do this explicitly or use the fact that

multiplying by **B** will transform vectors in V by $\mathcal{T}$. So we just multiply each column of **A** by **B** and store the results, in order, as columns in a new matrix **C**:

$$\mathbf{C} = \mathbf{BA}$$

If U has dimension n, V has dimension m, and W has dimension l, then **A** will be an $m \times n$ matrix and **B** will be an $l \times m$ matrix. Since the number of columns in **B** matches the number of rows in **A**, the matrix product can proceed, as we'd expect. The result **C** will be an $l \times n$ matrix and will apply the transformation of **A** followed by the transformation of **B** in a single matrix–vector multiplication.

This is the power of using matrices as a representation for linear transformations. By continually concatenating matrices, we can use the result to produce the effect of an entire series of transformations, in order, through a single matrix multiplication. Note that the order does matter. The preceding result **C** will perform the result of applying **A** followed by **B**. If we swap the terms (assuming they're still conformable under multiplication),

$$\mathbf{D} = \mathbf{AB}$$

and matrix **D** will perform the result of applying **B** followed by **A**. This is almost certainly not the same transformation.

For the discussion thus far, we have assumed that the resulting matrix will be applied to a vector represented as a column matrix. It is good to be aware that the choice of whether to represent a vector as a row matrix or column matrix affects the order of multiplications when combining matrices. Suppose we multiply a column vector **u** by three matrices, where the intended transformation order is to apply $\mathbf{M}_0$, then $\mathbf{M}_1$, and finally $\mathbf{M}_2$:

$$\begin{aligned} \mathbf{v} &= \mathbf{M}_0\mathbf{u} \\ \mathbf{w} &= \mathbf{M}_1\mathbf{v} \\ \mathbf{x} &= \mathbf{M}_2\mathbf{w} \end{aligned} \tag{3.6}$$

If we take equation 3.6 and substitute $\mathbf{M}_1\mathbf{v}$ for $\mathbf{w}$ and then $\mathbf{M}_0\mathbf{u}$ for $\mathbf{v}$, we get

$$\begin{aligned} \mathbf{x} &= \mathbf{M}_2\mathbf{M}_1\mathbf{v} \\ &= \mathbf{M}_2\mathbf{M}_1\mathbf{M}_0\mathbf{u} \\ &= \mathbf{M}_c\mathbf{u} \end{aligned}$$

Doing something similar for a row vector $\mathbf{a}^T$:

$$\begin{aligned}\mathbf{b}^T &= \mathbf{a}^T\mathbf{N}_0\\ \mathbf{c}^T &= \mathbf{b}^T\mathbf{N}_1\\ \mathbf{d}^T &= \mathbf{c}^T\mathbf{N}_2\end{aligned}$$

and substituting:

$$\begin{aligned}\mathbf{d}^T &= \mathbf{b}^T\mathbf{N}_1\mathbf{N}_2\\ &= \mathbf{a}^T\mathbf{N}_0\mathbf{N}_1\mathbf{N}_2\\ &= \mathbf{a}^T\mathbf{N}_r\end{aligned}$$

The order difference is quite clear. When using row vectors and concatenating, matrix order follows the left to right progress used in English text. Column vectors work right to left instead, which may not be as intuitive. We will just need to be careful about our matrix order and transpose any matrices that assume we're using row vectors.

There are two other ways to modify transformation matrices that aren't used as often. Instead of concatenating two transformations, we may want to create a new one by adding two together: $\mathcal{Q}(\mathbf{x}) = S(\mathbf{x}) + \mathcal{T}(\mathbf{x})$. This is easily done by adding the corresponding matrices together, so the matrix that performs $\mathcal{Q}$ is $\mathbf{C} = \mathbf{A} + \mathbf{B}$. Another means we might use for generating a new transformation from an existing one is to scale it: $\mathcal{R}(\mathbf{x}) = s \cdot \mathcal{T}(\mathbf{x})$. The corresponding matrix is created by scaling the original matrix: $\mathbf{D} = s\mathbf{A}$.

This concludes our main discussion of linear transformations and matrices. The remainder of the chapter will be concerned with other useful properties of matrices: solving systems of linear equations, determinants, and eigenvalues and eigenvectors.

3.4 Systems of Linear Equations

3.4.1 Definition

Other than performing linear transformations, another purpose of matrices is to act as a mechanism for solving systems of linear equations. A general system of m linear equations with n unknowns is represented as

$$\begin{aligned} b_0 &= a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} \\ b_1 &= a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} \\ &\vdots \quad \vdots \\ b_{m-1} &= a_{m-1,0}x_0 + a_{m-1,1}x_1 + \cdots + a_{m-1,n-1}x_{n-1} \end{aligned} \tag{3.7}$$

The problem we are trying to solve is: Given $a_{0,0}, \ldots, a_{m-1,n-1}$ and $b_0, \ldots, b_{m-1}$, what are the values of $x_0, \ldots, x_{n-1}$? For a given linear system, the set of all possible solutions is called the *solution set*.

As an example, the system of equations

$$\begin{aligned} x_0 + 2x_1 &= 1 \\ 3x_0 - x_1 &= 2 \end{aligned}$$

has the solution set $\{x_0 = 5/7, x_1 = 1/7\}$.

There may be more than one solution to the linear system. For example, the plane equation

$$ax + by + cz = -d$$

has an infinite number of solutions: The solution set for this example is all the points on the particular plane.

Alternatively, it may not be possible to find any solution to the linear system. Suppose that we have the linear system

$$\begin{aligned} x_0 + x_1 &= 1 \\ x_0 + x_1 &= 2 \end{aligned}$$

There are clearly no solutions for x and y. The solution set is the empty set.

Let's reexamine equation 3.7. If we think of $(x_0, \ldots, x_{n-1})$ as elements of an n-dimensional vector $\mathbf{x}$ and $(b_0, \ldots, b_{m-1})$ as elements of an m-dimensional vector $\mathbf{b}$, then this starts to look a lot like matrix multiplication. We can rewrite this as

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix}$$

or our old friend

$$\mathbf{Ax} = \mathbf{b}$$

The coefficients of the equation become the elements of matrix **A**, and matrix multiplication encapsulates our entire linear system. Now the problem becomes one of the form: Given **A** and **b**, what is **x**?

3.4.2 Solving Linear Systems

One case is very easy to solve. Suppose **A** looks like

$$\begin{bmatrix} 1 & a_{0,1} & \cdots & a_{0,n-1} \\ 0 & 1 & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

This is equivalent to the linear system

$$\begin{aligned} b_0 &= x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} \\ b_1 &= x_1 + \cdots + a_{1,n-1}x_{n-1} \\ &\vdots \quad \vdots \\ b_{m-1} &= x_{n-1} \end{aligned}$$

We see that we immediately have the solution to one unknown via $x_{n-1} = b_{m-1}$. We can substitute this value into the previous $m - 1$ equations and possibly solve for another x_i. If so, we can substitute that x_i into the remaining unsolved equations and so on up the chain. If there is a single solution for the system of equations, we will find it; otherwise, we will solve as many terms as possible and derive a solution set for the remainder.

This matrix is said to be in *row echelon form*. The formal definition for row echelon form is

1. If a row is entirely zeros, it will be below any nonzero rows of the matrix; in other words, all zero rows will be at the bottom of the matrix.
2. The first nonzero element of a row (if any) will be 1 (called a *leading 1*).
3. Each leading 1 will be to the right of a leading 1 in any preceding row.

If the following additional condition is met, we say that the matrix is in *reduced row echelon form*.

4. Each column with a leading 1 will be zero in the other rows.

The process we've described gives us a clue about how to proceed in solving general systems of linear equations. Suppose we can multiply both sides of our equation by a series of matrices so that the left-hand side becomes a matrix in row echelon form. Then we can use this in combination with the right-hand side to give us the solution for our system of equations.

However, we need to use matrices that preserve the properties of the linear system; the solution set for both systems of equations must remain equal. This restricts us to those matrices that perform one of three transformations called *elementary row operations*. These are

1. Multiply a row by a nonzero scalar.
2. Add a nonzero multiple of one row to another.
3. Swap two rows.

These three types of transformations maintain the solution set of the linear system while allowing us to reduce it to a simpler problem. The matrices that perform elementary row operations are called *elementary matrices*.

Some simple examples of elementary matrices include one that multiplies row 2 by a scalar a:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

one that adds k times row 2 to row 1:

$$\begin{bmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and one that swaps rows 2 and 3:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

3.4.3 Gaussian Elimination

SOURCE CODE LIBRARY IvMath FILENAME IvGaussianElim

In practice we don't solve linear systems through matrix multiplication. Instead, it is more efficient to iteratively perform the operations directly on **A** and **b**. The most basic method for solving linear systems is known as *Gaussian*

elimination, after Karl Friedrich Gauss, a prolific German mathematician of the eighteenth and nineteenth centuries. It involves concatenating the matrix **A** and vector **b** into a form called an *augmented matrix* and then performing a series of elementary row operations on the augmented matrix, in a particular order. This will either give us a solution to the system of linear equations or tell us that computing a single solution is not possible; that is, either there is no solution or an infinite number of solutions.

To create the augmented matrix, we take the original matrix **A** and combine it with our constant vector **b**, for example,

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 3 \\ 4 & 5 & 6 & 2 \\ 7 & 8 & 9 & 1 \end{array}\right]$$

The vertical line within the matrix indicates the separation between **A** and **b**. To this augmented matrix, we will directly apply one or more of our row operations.

The process begins by looking at the first element in the first row. The first step is called a *pivoting* step. At the very least we need to ensure that we have a nonzero entry in the diagonal position, so if necessary we will swap this row with one of the lower rows with a nonzero entry in the same column. The element that we're swapping into place is called the *pivot* element, and swapping two rows to move the pivot element into place is known as *partial pivoting*. For better numerical precision, we usually go one step further and swap with the row that contains the element of largest absolute value. If no pivot element can be found, then there is no single solution and we abort.

Now let's say that the current pivot element value is k. We scale the entry row by $1/k$ to set the diagonal entry to 1. Finally, we set the column elements below the diagonal entry to zero by adding appropriate multiples of the current row. Then we move on to the next row and look at its diagonal entry. At the end of this process, our matrix will be in row echelon form.

Let's take a look at an example. Suppose we have the following system of linear equations:

$$\begin{array}{rrcrl} x & -3y & + & z & = 5 \\ 2x & -y & + & 2z & = 5 \\ 3x & +6y & + & 9z & = 3 \end{array}$$

The equivalent augmented matrix is

$$\left[\begin{array}{ccc|c} 1 & -3 & 1 & 5 \\ 2 & -1 & 2 & 5 \\ 3 & 6 & 9 & 3 \end{array}\right]$$

If we look at column 0, the maximal entry is 3, in row 2. So we begin by swapping row 2 with row 0:

$$\left[\begin{array}{ccc|c} 3 & 6 & 9 & 3 \\ 2 & -1 & 2 & 5 \\ 1 & -3 & 1 & 5 \end{array}\right]$$

We scale the new row 0 by 1/3 to set the pivot element to 1:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 2 & -1 & 2 & 5 \\ 1 & -3 & 1 & 5 \end{array}\right]$$

Now we start clearing the lower entries. The first entry in row 1 is 2, so we scale row 0 by -2 and add it to row 1:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -5 & -4 & 3 \\ 1 & -3 & 1 & 5 \end{array}\right]$$

We do the same for row 2, scaling by -1 and adding:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & -5 & -4 & 3 \\ 0 & -5 & -2 & 4 \end{array}\right]$$

We are done with row 0 and move on to row 1. Row 1, column 1, is the maximal entry in the column, so we don't need to swap rows. However, it isn't 1, so we need to scale row 1 by $-1/5$:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 4/5 & -3/5 \\ 0 & -5 & -2 & 4 \end{array}\right]$$

We now need to clear element 1 of row 2 by scaling row 1 by 5 and adding:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 4/5 & -3/5 \\ 0 & 0 & 2 & 1 \end{array}\right]$$

Finally, we scale the bottom row by 1/2 to set the pivot element in the row to 1:

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 0 & 1 & 4/5 & -3/5 \\ 0 & 0 & 1 & 1/2 \end{array}\right]$$

This matrix is now in row echelon form. We have two possibilities at this point. We could clear the upper triangle of the matrix in a fashion similar to how we cleared the lower triangle, but by working up from the bottom and adding multiples of rows. The solution $\mathbf{x}$ to the linear system would end up in the right-hand column. This is known as *Gauss-Jordan elimination*.

But let's look at the linear system we have now:

$$\begin{aligned} x + 2y + 3z &= 1 \\ y + 4/5z &= -3/5 \\ z &= 1/2 \end{aligned}$$

As expected, we already have a known quantity: z. If we plug z into the second equation, we can solve for y:

$$\begin{aligned} y &= -3/5 - 4/5z && (3.8) \\ &= -3/5 - 4/5(1/2) && (3.9) \\ &= -1 && (3.10) \end{aligned}$$

Once y is known, we can solve for x:

$$\begin{aligned} x &= 1 - 2y - 3z && (3.11) \\ &= 1 - 2(-1) - 3(1/2) && (3.12) \\ &= 3/2 && (3.13) \end{aligned}$$

So our final solution for $\mathbf{x}$ is $(3/2, -1, 1/2)$.

This process of substituting known quantities into our equations is called *back substitution*.

A summary of Gaussian elimination with back substitution follows:

```
for p = 1 to n do
    // find the element with largest absolute value in col p

    // if max is zero, stop!

    // if max element not in row p, swap rows

    // set pivot element to 1
    multiply row p by 1/A[p][p]

    // clear lower column entries
    for r = p+1 to n do
```

```
            subtract row p times A[r,p] from current row,
                so that element in pivot column becomes 0

    // do backwards substitution
    for row = n-1 to 1
        for col = row+1 to n
            // subtract out known quantities
            b[row] = b[row] - A[row][col]*b[col]
```

The pseudocode shows what may happen when we encounter a linear system with no single solution. If we can't swap a nonzero entry in the pivot location, then there is a column that is all zeros. This is only possible if the rank of the matrix (i.e., the number of linearly independent column vectors) is less than the number of unknowns. In this case, there is no solution to the linear system and we abort.

In general, we can state that if the rank of the coefficient matrix **A** equals the rank of the augmented matrix **A|b**, then there will be at least one solution to the linear system. If the two ranks are unequal, then there are no solutions. There is a single solution only if the rank of **A** is equal to the minimum of the number of rows or columns of **A**.

3.5 Matrix Inverse

This may seem like a lot of trouble to go to solve a simple equation like $\mathbf{b} = \mathbf{A}\mathbf{x}$. If this were scalar math, we could simply divide both sides of the equation by **A** to get

$$\mathbf{x} = \mathbf{b}/\mathbf{A}$$

Unfortunately, matrices don't have a division operation. However, we can use an equivalent concept: the inverse.

3.5.1 Definition

In scalar multiplication, the inverse is defined as the reciprocal:

$$x \cdot \frac{1}{x} = 1$$

or

$$x \cdot x^{-1} = 1$$

Correspondingly, for a given matrix $\mathbf{A}$, we can define its inverse $\mathbf{A}^{-1}$ as a matrix such that

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}$$

and

$$\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I}$$

There are a few things that fall out from this definition. First of all, in order for the first multiplication to occur, the number of rows in the inverse must be the same as the number of columns in the original matrix. For the second to occur, the converse is true. So, the matrix and its inverse must be square and the same size. Since not all matrices are square, it's clear that not every matrix has an inverse.

Second, the inverse of the inverse returns the original matrix. Given

$$\mathbf{A}^{-1} \cdot (\mathbf{A}^{-1})^{-1} = \mathbf{I}$$

and

$$\mathbf{A}^{-1} \cdot \mathbf{A} = I$$

then

$$(\mathbf{A}^{-1})^{-1} = \mathbf{A}$$

Even if a matrix is square, there isn't always an inverse. An extreme example is the zero matrix. Any matrix multiplied by this gives the zero matrix, so there is no matrix multiplication that will produce the identity. Another set of examples is matrices that have a zero row or column vector. Multiplying by such a row or column will return a dot product of zero, so you'll end up with a zero row or column vector in the product as well—again, not the identity matrix. In general, if the null space of the matrix is nonzero, then the matrix is noninvertible; that is, the matrix is only invertible if the rank of the matrix is equal to the number of rows and columns.

Given these identities, we can now solve for our preceding linear system. Recall that the equation was

$$\mathbf{Ax} = \mathbf{b}$$

If we multiply both sides by $\mathbf{A}^{-1}$, then

$$\begin{aligned}\mathbf{A}^{-1}\mathbf{A}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b}\\ \mathbf{I}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b}\\ \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b}\end{aligned}$$

Therefore, if we could find the inverse of $\mathbf{A}$, we could use it to solve for $\mathbf{x}$. This is not usually a good idea, computationally speaking. It's usually cheaper to solve for $\mathbf{x}$ directly, rather than generating the inverse and then performing the matrix multiplication. The latter can also lead to increased numerical error. However, sometimes finding the inverse is a necessary evil.

The left-hand side of the above derivation shows us that we can think of the inverse $\mathbf{A}^{-1}$ as undoing the effect of $\mathbf{A}$. If we start with $\mathbf{A}\mathbf{x}$ and premultiply by $\mathbf{A}^{-1}$, we get back $\mathbf{x}$, our original vector.

We can find the inverse of a matrix using Gaussian elimination to solve for it column by column. Suppose we call the first column of $\mathbf{A}^{-1}$ $\mathbf{x}_0$. We can represent this as

$$\mathbf{x}_0 = \mathbf{A}^{-1}\mathbf{e}_0$$

where, as we recall, $\mathbf{e}_0 = (1, 0, \ldots, 0)$. Multiplying both sides by $\mathbf{A}$ gives

$$\mathbf{A}\mathbf{x}_0 = \mathbf{e}_0$$

Finding the solution to this linear system gives us the first column of $\mathbf{A}^{-1}$. We can do the same for the other columns, but using $\mathbf{e}_1$, $\mathbf{e}_2$, and so on. Instead of solving these one at a time, though, it is more efficient to create an augmented matrix with $\mathbf{A}$ and $\mathbf{e}_0, \ldots, \mathbf{e}_{n-1}$ as columns on the right, or just $\mathbf{I}$. For example,

$$\left[\begin{array}{ccc|ccc} 2 & 0 & 4 & 1 & 0 & 0\\ 0 & 3 & -9 & 0 & 1 & 0\\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

If we use Gauss-Jordan elimination to turn the left-hand side of the augmented matrix into the identity matrix, then we will end up with the inverse (if any) on the right-hand side. From here we perform our elementary row operations as before. The maximal entry is already in the pivot point, so we scale the first row by 1/2:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2 & 1/2 & 0 & 0\\ 0 & 3 & -9 & 0 & 1 & 0\\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

The nonpivot entries in the first column are zero, so we move to the second column. Scaling the second row by 1/3 to set the pivot point to 1 gives us

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 2 & 1/2 & 0 & 0 \\ 0 & 1 & -3 & 0 & 1/3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Again, our nonpivot entries in the second column are 0, so we move to the third column. Our pivot entry is 1, so we don't need to scale. We add -2 times the last row to the first row to clear that entry, then 3 times the last row to the second row to clear that entry, and get

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1/2 & 0 & -2 \\ 0 & 1 & 0 & 0 & 1/3 & 3 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

The inverse of our original matrix is now on the right-hand side of the augmented matrix.

3.5.2 Simple Inverses

Gaussian elimination, while useful, is unnecessary for computing the inverse of many of the matrices we will be using. The majority of matrices that we will encounter in games and three-dimensional (3D) applications have simple inverses, and knowing the form of the matrix can make computing the inverse trivial.

One case is that of an *orthogonal matrix*, where the component row or column vectors are orthonormal. Recall that this means that the vectors are of unit length and perpendicular. If a matrix $\mathbf{A}$ is orthogonal, its inverse is the transpose:

$$\mathbf{A}^{-1} = \mathbf{A}^T$$

One example of an orthogonal matrix is

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Another simple case is a diagonal matrix with nonzero elements in the diagonal. The inverse of such a matrix is also diagonal, where the new diagonal

elements are the reciprocal of the original diagonal elements, as shown by the following:

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}^{-1} = \begin{bmatrix} 1/a & 0 & 0 \\ 0 & 1/b & 0 \\ 0 & 0 & 1/c \end{bmatrix}$$

The third case is a modified identity matrix, where the diagonal is all 1s but one column or row is nonzero. One such 3×3 matrix is

$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}$$

For a matrix of this form, we simply negate the nonzero elements to invert it. Using the previous example,

$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, we can combine this knowledge to take advantage of an algebraic property of matrices. If we have two square matrices **A** and **B**, both of which are invertible, then

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

So, if we know that our current matrix is the product of any of the cases we've just discussed, we can easily compute its inverse using the preceding formula. This will prove to be useful in subsequent chapters.

3.6 Determinant

3.6.1 Definition

The *determinant* is a scalar quantity created by evaluating the elements of a square matrix. In real vector spaces, it acts as a general measure of how vectors transformed by the matrix change in size. For example, if we take the columns of a 2×2 matrix (i.e., the transformed basis vectors) and use them as the sides of a parallelogram (Figure 3.2), then the absolute value of the determinant is equal to the area of a parallelogram. For a 3×3 matrix, the

absolute value of the determinant is equal to the volume of a parallelepiped described by the three transformed basis vectors (Figure 3.3).

The sign of the determinant depends on whether or not we have switched our ordered basis vectors from being relatively right-handed to being left-handed. In Figure 3.2, the shortest angle from $\mathbf{a}_0$ to $\mathbf{a}_1$ is clockwise, so they are left-handed. The determinant, therefore, is negative.

We represent the determinant in one of two ways, either $\det(\mathbf{A})$ or $|\mathbf{A}|$. The first is more often used with a symbol, and the second when showing the elements of a matrix:

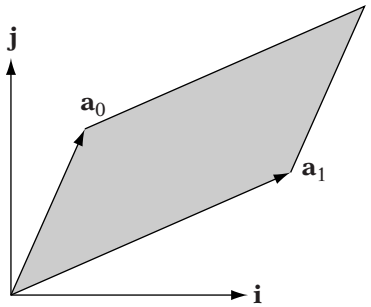


FIGURE 3.2 Determinant of 2×2 matrix as area of parallelogram bounded by transformed basis vectors $\mathbf{a}_0$ and $\mathbf{a}_1$.

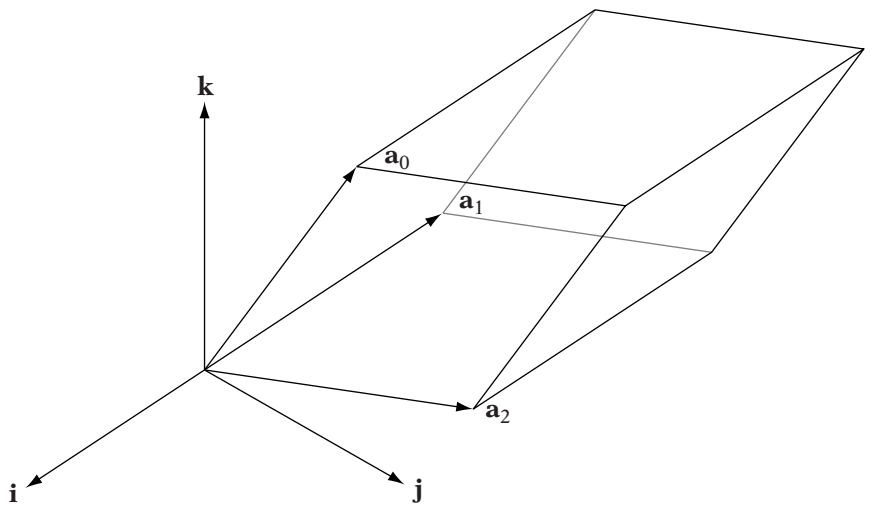


FIGURE 3.3 Determinant of 3×3 matrix as volume of parallelepiped bounded by transformed basis vectors $\mathbf{a}_0$, $\mathbf{a}_1$, and $\mathbf{a}_2$.

$$\det(\mathbf{A}) = \begin{vmatrix} 1 & -3 & 1 \\ 2 & -1 & 2 \\ 3 & 6 & 9 \end{vmatrix}$$

The diagrams showing area of a parallelogram and volume of a parallelepiped should look familiar from our discussion of cross product and triple scalar product. In fact, the cross product is sometimes represented as

$$\mathbf{v} \times \mathbf{w} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$$

while the triple product is represented as

$$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = \begin{vmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$$

Since $\det(\mathbf{A}^T) = \det(\mathbf{A})$, this representation is equivalent.

3.6.2 Computing the Determinant

There are a few ways of representing the determinant computation for a specific matrix $\mathbf{A}$. A standard recursive definition, choosing any row i, is

$$\det(\mathbf{A}) = \sum_{j=1}^{n} a_{i,j}(-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$$

Alternatively, we can expand by column j instead:

$$\det(\mathbf{A}) = \sum_{i=1}^{n} a_{i,j}(-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$$

In both cases, $\tilde{\mathbf{A}}_{i,j}$ is the submatrix formed by removing the ith row and jth column from $\mathbf{A}$. The base case is the determinant of a matrix with a single element, which is the element itself.

The term $\det(\tilde{\mathbf{A}}_{i,j})$ is also referred to as the *minor of entry* $a_{i,j}$, and the term $(-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$ is called the *cofactor of entry* $a_{i,j}$.

The first formula tells us that for a given row i, we multiply each row entry $a_{i,j}$ by the determinant of the submatrix formed by removing row i and column j and either add or subtract it to the total depending on its position

in the matrix. The second does the same but moves along column j instead of row i.

Let's compute an example determinant, expanding by row 0:

$$\det\left(\begin{bmatrix} 1 & 1 & 2 \\ 2 & 4 & -3 \\ 3 & 6 & -5 \end{bmatrix}\right) = ?$$

The first element of row 0 is 1, and the submatrix with row 0 and column 0 removed is

$$\begin{bmatrix} 4 & -3 \\ 6 & -5 \end{bmatrix}$$

The second element is also 1. However, we negate it since we are considering row 0 and column 1: $0 + 1 = 1$, which is odd. The submatrix is $\mathbf{A}$ with row 0 and column 1 removed:

$$\begin{bmatrix} 2 & -3 \\ 3 & -5 \end{bmatrix}$$

The third element of the row is 2, with the submatrix

$$\begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix}$$

We don't negate it since we are considering row 0 and column 2: $0 + 2 = 2$, which is even.

So, the determinant is

$$\begin{aligned} \det(\mathbf{A}) &= 1 \cdot \begin{vmatrix} 4 & -3 \\ 6 & -5 \end{vmatrix} - 1 \cdot \begin{vmatrix} 2 & -3 \\ 3 & -5 \end{vmatrix} + 2 \cdot \begin{vmatrix} 2 & 4 \\ 3 & 6 \end{vmatrix} \\ &= -1 \end{aligned}$$

In general, the determinant of a 2×2 matrix is

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = a \cdot \det([d]) - b \cdot \det([c]) = ad - bc$$

And the determinant of a 3×3 matrix is

$$\det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) = a \cdot \det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b \cdot \det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c \cdot \det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right)$$

or

$$a(ei - fh) - b(di - fg) + c(dh - eg)$$

There are some additional properties of the determinant that will be useful to us. If we have two $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$, the following hold:

1. $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$.
2. $\det(\mathbf{A}^{-1}) = \dfrac{1}{\det(\mathbf{A})}$.

We can look at the value of the determinant to tell us some features of our matrix. First of all, as we have mentioned, any matrix that transforms our basis vectors from right-handed to left-handed will have a negative determinant. If the matrix is also orthogonal, we call a matrix of this type a *reflection*. We will learn more about reflection matrices in the next chapter.

Then there are matrices that have a determinant of 1. The matrices we will encounter most often with this property are orthogonal matrices, where the handedness of the resulting basis stays the same (i.e., a right-handed basis is transformed to a right-handed basis). Figure 3.4 provides an example. Our transformed basis vectors are $(-\sqrt{2}/2, \sqrt{2}/2)$ and $(\sqrt{2}/2, \sqrt{2}/2)$. They remain orthonormal, so their area is just the product of the lengths of the two vectors, or 1×1 or 1. This type of matrix is called a *rotation*. As with reflections, we'll see more of rotations in the next chapter.

Finally, if the determinant is 0, then we know that the matrix has no inverse. The obvious case is if the matrix has a row or column of all 0s. Look again at our formula for the determinant. Suppose row i is all 0s. Multiplying all the submatrices against this row and summing together will clearly give us 0 as a result. The same is true for a zero column. The other and related possibility is that we have a linearly dependent row or column vector. In both cases the rank of the matrix is less than n — the size of the matrix — and therefore the matrix does not have an inverse. So, if the determinant of a matrix is 0, we know the matrix is not invertible.

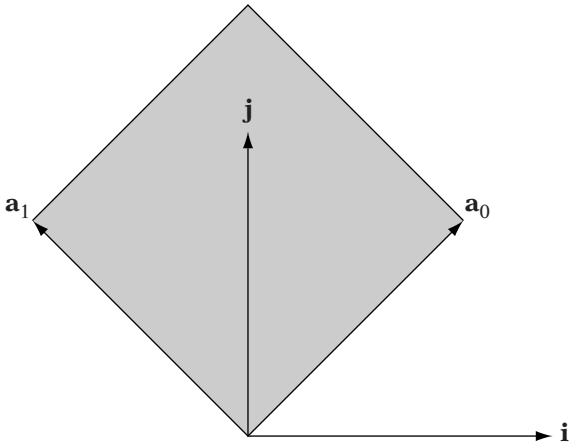


FIGURE 3.4 Determinant of example 2×2 orthogonal matrix.

3.6.3 Determinants and Elementary Row Operations

SOURCE CODE
LIBRARY
lvMath
FILENAME
lvGaussianElim

For 2×2 and 3×3 matrices, computing the determinant in this manner is a simple process. However, for larger and larger matrices, our recursive definition becomes unwieldy, and for large enough n, will take an unreasonable amount of time to compute. In addition, computing the determinant in this manner can lead to floating-point precision problems. Fortunately, there is another way.

Suppose we have an upper triangular matrix $\mathbf{U}$. The first part of the determinant sum is $u_{0,0}\tilde{\mathbf{U}}_{0,0}$. The other terms, however, are 0, because the first column with the first row removed is all 0s. So the determinant is just

$$\det(\mathbf{U}) = u_{0,0}\tilde{\mathbf{U}}_{0,0}$$

If we expand the recursion, we find that the determinant is the product of all the diagonal elements, or

$$\det(\mathbf{U}) = u_{0,0}u_{1,1}\ldots u_{nn}$$

As we did when solving linear systems, we can use Gaussian elimination to change our matrix into row echelon form, which is an upper triangular matrix. However, this assumes that elementary row operations have no effect on the determinant, which is not the case. Let's look at a few examples.

Suppose we have the matrix

$$\begin{bmatrix} 2 & -4 \\ -1 & 1 \end{bmatrix}$$

The determinant of this matrix is -2. If we multiply the first row by $1/2$, we get

$$\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix}$$

which has a determinant of -1. Multiplying a row by a scalar k multiplies the determinant by k as well.

Now suppose we add two times the first row to the second one. We get

$$\begin{bmatrix} 1 & -2 \\ 1 & -3 \end{bmatrix}$$

which also has a determinant of -1. Adding a multiple of one row to another has no effect on the determinant.

Finally, we can swap row 1 with row 2:

$$\begin{bmatrix} 1 & -3 \\ 1 & -2 \end{bmatrix}$$

which has a determinant of 1. Swapping two rows or two columns changes the sign of the determinant.

The effect of elementary row operations on the determinant can be summarized as follows:

Multiply row by k:	Multiplies determinant by k
Add multiple of one row to another:	No effect
Swap rows:	Changes sign of determinant

Therefore, our approach for calculating the determinant for a general matrix is this: As we perform Gaussian elimination, we keep a running product p of any multiplies we do to create leading 1s and negate p for every row swap. If we find a zero column when we look for a pivot element, we know the determinant is 0 and return such.

Let's suppose our final product is p. This represents what we've multiplied the determinant of our original matrix by to get the determinant of the final matrix $\mathbf{A}'$, or

$$p \cdot \det(\mathbf{A}) = \det(\mathbf{A}')$$

so

$$\det(\mathbf{A}) = \frac{1}{p} \cdot \det(\mathbf{A}')$$

We know that the determinant of $\mathbf{A}'$ is 1, since the diagonal of the row echelon matrix is all 1s. So our final determinant is just $1/p$. However, this is just the product of the multiplies we do to create leading 1s, and -1 for every row swap, or

$$p = \frac{1}{p_{0,0}} \frac{1}{p_{1,1}} \cdots \frac{1}{p_{n,n}} (-1)^k$$

where k is the number of row swaps. Then,

$$1/p = p_{0,0} p_{1,1} \cdots p_{n,n} (-1)^k$$

So all we need to do is multiply our running product by each pivot element and negate for each row swap. At the end of our Gaussian elimination process, our running product will be the determinant we seek.

3.6.4 Adjoint Matrix and Inverse

Recall that the cofactor of an entry $a_{i,j}$ is

$$C_{i,j} = (-1)^{(i+j)} \det(\tilde{\mathbf{A}}_{i,j})$$

For an $n \times n$ matrix, we can construct a corresponding matrix where we replace each element with its corresponding cofactor, or

$$\begin{bmatrix} C_{0,0} & C_{0,1} & \cdots & C_{0,n-1} \\ C_{1,0} & C_{1,1} & \cdots & C_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n-1,1} & C_{n-1,2} & \cdots & C_{n-1,n-1} \end{bmatrix}$$

This is called the matrix of cofactors from $\mathbf{A}$, and its transpose is the *adjoint matrix* $\mathbf{A}^{\text{adj}}$.

Gabriel Cramer, a Swiss mathematician, showed that the inverse of a matrix can be computed from the adjoint by

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \mathbf{A}^{\text{adj}}$$

Many graphics engines use *Cramer's method* to compute the inverse, and for 3×3 and 4×4 matrices it's not a bad choice; for matrices of this size, Cramer's method is actually faster than Gaussian elimination. Because of this, we have chosen to implement `IvMatrix33::Inverse()` using an efficient form of Cramer's method.

However, whether you're using Gaussian elimination or Cramer's method, you're probably doing more work than is necessary for the matrices we will encounter. Most will be in one of the formats described in Section 3.5.2 or a multiple of these matrix types. Using the process described in that section, you can compute the inverse by decomposing the matrix into a set of these types, inverting the simple matrices, and multiplying in reverse order to compute the matrix. This is often faster than either Gaussian elimination or Cramer's method and can be more tolerant of floating-point errors because you can find near-exact solutions for the simple matrices.

3.7 Eigenvalues and Eigenvectors

There are two more properties of a matrix that we can find useful in certain circumstances: the *eigenvalue* and *eigenvector*. If we have an $n \times n$ matrix $\mathbf{A}$, then a nonzero vector $\mathbf{x}$ is called an eigenvector if there is some scalar value λ such that

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{3.14}$$

In this case, the value λ is the eigenvalue associated with that eigenvector.

We can solve for the eigenvalues of a matrix by rewriting equation 3.14 as

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{I}\mathbf{x} \tag{3.15}$$

or

$$(\lambda\mathbf{I} - \mathbf{A})\mathbf{x} = \mathbf{0}$$

It can be shown that there is a nonzero solution of this equation if and only if

$$\det(\lambda\mathbf{I} - \mathbf{A}) = 0$$

This is called the *characteristic equation* of $\mathbf{A}$. Expanding this equation gives us an n-degree polynomial of λ, and solving for the roots of this equation will give us the eigenvalues of the matrix.

Now, for a given eigenvalue there will be an infinite number of associated eigenvectors, all scalar multiples of each other. This is called the *eigenspace* for that eigenvalue. To find the eigenspace for a particular eigenvector, we simply substitute that eigenvalue into equation 3.15 and solve for **x**.

In practice, solving the characteristic equation becomes more and more difficult the larger the matrix. However, there is a particular class of matrices called *real symmetric matrices*, so called because they only have real elements and are diagonally symmetric. Such matrices have a few nice properties. First of all, their eigenvectors are orthogonal. Secondly, it is possible to find a matrix **R**, such that $\mathbf{R}^T\mathbf{AR}$ is a diagonal matrix **D**. It turns out that the columns of **R** are the eigenvectors of **A**, and the diagonal elements of **D** are the corresponding eigenvectors. This process is called *diagonalization*.

There are a number of standard methods for finding **R**. One such is the Jacobi method, which computes a series of matrices to iteratively diagonalize **A**. These matrices are then concatenated to create **R**. The problem with this method is that it is not always guaranteed to converge to a solution. An alternative is the Householder-QR/QL method, which similarly computes a series of matrices, but this time the end result is a tridiagonal matrix. From this we can perform a series of steps that factor the matrix into an orthogonal matrix **Q** and upper triangular matrix **R** (or an orthogonal matrix **Q** and a lower triangular matrix **L**). This will eventually diagonalize the matrix, again allowing us to compute the eigenvectors and eigenvalues. This can take more steps than the Jacobi method, but is guaranteed to complete in a fixed amount of time.

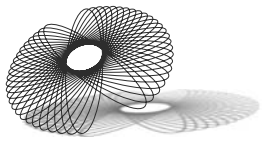
For 3×3 real symmetric matrices, Eberly [28] has a method that solves for the roots of the characteristic equation. This is considerably more efficient than the Householder method, and is relatively straightforward to compute.

3.8 Chapter Summary

In this chapter, we've discussed the general properties of linear transformations and how they are represented and performed by matrices. Matrices also can be used to compute solutions to linear systems of equations by using either Gaussian elimination or similar methods. We covered some basic matrix properties, the concepts of matrix identity and inverse (and various methods for calculating the latter), and the meaning and calculation of the determinant. This lays the foundation for what we'll be discussing in the next chapter: Using matrix transformations to manipulate models in a 3D world.

For those who are interested in reading further, Anton and Rorres [3] is a standard reference for many first courses in linear algebra. Other texts with slightly different approaches include Axler [4] and Friedberg et al. [39]. More information on Gaussian elimination and its extensions, such as LU decomposition, can be found in Anton and Rorres [3] as well as in the *Numerical Recipes* series [96]. Finally, Blinn has an excellent article in his collection *Notation, Notation, Notation* [9] on the geometry underlying 2×2 matrix operations.

This page intentionally left blank

Chapter 4

Affine Transformations

4.1 Introduction

Now that we've chosen a mathematically sound basis for representing geometry in our game and discussed some aspects of matrix arithmetic, we need to combine them into an efficient method for placing and moving virtual objects or models. There are a few reasons we seek this efficiency. Suppose we wish to build a core level in our game space, say the office of a computer company. We could build all of our geometry in place and hard-code all of the locations. However, if we have a number of objects that are duplicated throughout the space — computers, desks, and chairs, for example — it would be more memory-efficient to create one master copy of the geometry for each type of object. Then, for each instance of a particular object, we can specify just a position and orientation and let the rendering and simulation engine handle the placement.

Another, more obvious reason is that objects in games generally move so that setting them at a fixed location is not practical. We will need to have some means to specify, for a model as a whole, its position and orientation in space.

There are a few characteristics we desire in our method. We want it to be fast and work well with our existing data and math library. We want to be able to concatenate a series of operations so we can perform them with a single operation, just as we did with linear transformations. Since our objects consist of collections of points, we need our method to work on points in an affine space, but we'll still need to transform vectors as well. The specific method we will use is called an *affine transformation*.

4.2 Affine Transformations

4.2.1 Matrix Definition

In the last chapter we discussed linear transformations, which map from one vector space to another. We can apply such transformations to vectors using matrix operations. There is a nearly equivalent set of transformations that map between affine spaces, which we can apply to points and vectors in an affine space. These are known as *affine transformations* and they too can be applied using matrix operations, albeit in a slightly different form.

In the simplest terms, an affine transformation on a point can be represented by a matrix multiplication followed by a vector add, or,

$$\mathbf{A}\mathbf{x} + \mathbf{y}$$

where the matrix $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{y}$ is an m-vector, and $\mathbf{x}$ consists of the point coordinates $(x_0, \ldots, x_{n-1})$.

We can represent this process of transformation by using block matrices:

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{A}\mathbf{x} + \mathbf{y} \\ 1 \end{bmatrix} \tag{4.1}$$

As we can see, in order to allow the multiplication to proceed, we'll represent our point with a trailing 1 component. However, for the purposes of computation, the vector $\mathbf{0}^T$, the 1 in the lower right-hand corner of the matrix, and the trailing 1s in the points are unnecessary. They take up memory and using the full matrix takes additional instructions to multiply by constant values. Because of this, an affine transformation matrix is sometimes represented in a form where these constant terms are implied, either as an $m \times (n + 1)$ matrix or as the matrix multiplication plus vector add form above.

If we subtract two points in an affine space, we get a vector:

$$\begin{aligned} \mathbf{v} &= P_0 - P_1 \\ &= \begin{bmatrix} \mathbf{x}_0 \\ 1 \end{bmatrix} - \begin{bmatrix} \mathbf{x}_1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_0 - \mathbf{x}_1 \\ 0 \end{bmatrix} \end{aligned}$$

As we can see, a vector is represented in an affine space with a trailing 0. As previously noted in Chapter 2, this provides justification for some math

libraries to use the trailing 1 on points and trailing 0 on vectors. If we multiply a vector using this representation by our $(m+1) \times (n+1)$ matrix,

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} \mathbf{A}\,\mathbf{v} \\ 0 \end{bmatrix}$$

we see that the vector is affected by the upper left $m \times n$ matrix $\mathbf{A}$, but not the vector $\mathbf{y}$. This has the same effect on the first n elements of $\mathbf{v}$ as multiplying an n-dimensional vector by $\mathbf{A}$, which is a linear transformation. So, this representation allows us to use affine transformation matrices to apply linear transformations on vectors in an affine space.

Suppose we wish to concatenate two affine transformations $\mathcal{S}$ and $\mathcal{T}$, where the matrix representing $\mathcal{S}$ is

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

and the matrix representing $\mathcal{T}$ is

$$\begin{bmatrix} \mathbf{B} & \mathbf{z} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

As with linear transformations, to find the matrix that represents the composition of $\mathcal{S}$ and $\mathcal{T}$, we multiply the matrices together. This gives

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{B} & \mathbf{z} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{AB} & \mathbf{A}\,\mathbf{z} + \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.2}$$

Finding the inverse for an affine transformation is equally as straightforward. Again, we can use a process similar to the one we used with linear transformation matrices. Starting with

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{0}^T & 1 \end{bmatrix}$$

we multiply by both sides to remove the $\mathbf{y}$ component from the left-most matrix:

$$\begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{0}^T & 1 \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

We then multiply by both sides to change the left-most matrix to the identity:

$$\begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

thereby giving us the inverse on the right-hand side.

When we're working in $\mathbb{R}^3$, $\mathbf{A}$ will be a 3×3 matrix and $\mathbf{y}$ will be a 3-vector; hence the full affine matrix will be a 4×4 matrix. Most graphics libraries expect transformations to be in the 4×4 matrix form, so if we do use the more compact forms in our math library to save memory, we will still have to expand them before rendering our objects. Because of this, we will use the 4×4 form for our following discussions, with the understanding that in our ultimate implementation we may choose one of the other forms for efficiency's sake.

4.2.2 Formal Definition

While the definition above will work for most practical purposes, to truly understand what our matrix form does requires some further explanation. We'll begin by formally defining an affine transformation. Recall that linear transformations preserve the linear operations of vector addition and scalar multiplication. In other words, linear transformations map from one vector space to another and preserve linear combinations. Thus, for a given linear transformation $\mathcal{S}$:

$$\mathcal{S}(a_0\mathbf{v}_0 + a_1\mathbf{v}_1 + \cdots + a_{n-1}\mathbf{v}_{n-1}) = a_0\mathcal{S}(\mathbf{v}_0) + a_1\mathcal{S}(\mathbf{v}_1) + \cdots + a_{n-1}\mathcal{S}(\mathbf{v}_{n-1})$$

Correspondingly, an affine transformation $\mathcal{T}$ maps between two affine spaces A and B and preserves affine combinations. For scalars $a_0, \ldots, a_{n-1}$ and points $P_0, \ldots, P_{n-1}$ in A:

$$\mathcal{T}(a_0P_0 + \cdots + a_{n-1}P_{n-1}) = a_0\mathcal{T}(P_0) + \cdots + a_{n-1}\mathcal{T}(P_{n-1})$$

where $a_0 + \cdots + a_{n-1} = 1$.

As with our test for linear transformations, to determine whether a given transformation $\mathcal{T}$ is an affine transformation, it is sufficient to test a single affine combination:

$$\mathcal{T}(a_0P_0 + a_1P_1) = a_0\mathcal{T}(P_0) + a_1\mathcal{T}(P_1)$$

where $a_0 + a_1 = 1$.

Affine transformations are particularly useful to us because they preserve certain properties of geometry. First, they maintain collinearity, so points on a line will remain collinear and points on a plane will remain coplanar when transformed.

If we transform a line:

$$\begin{aligned} L(t) &= (1-t)P_0 + tP_1 \\ \mathcal{T}(L(t)) &= \mathcal{T}((1-t)P_0 + tP_1) \\ &= (1-t)\mathcal{T}(P_0) + t\mathcal{T}(P_1) \end{aligned}$$

the result is clearly still a line (assuming $\mathcal{T}(P_0)$ and $\mathcal{T}(P_1)$ aren't coincident). Similarly, if we transform a plane:

$$\begin{aligned} P(t) &= (1-s-t)P_0 + sP_1 + tP_2 \\ \mathcal{T}(P(t)) &= \mathcal{T}((1-s-t)P_0 + sP_1 + tP_2) \\ &= (1-s-t)\mathcal{T}(P_0) + s\,\mathcal{T}(P_1) + t\,\mathcal{T}(P_2) \end{aligned}$$

the result is clearly a plane (assuming $\mathcal{T}(P_0)$, $\mathcal{T}(P_1)$, and $\mathcal{T}(P_2)$ aren't collinear).

The second property of affine transformations is that they preserve relative proportions. The point that lies at t distance between P_0 and P_1 on the original line will map to the point that lies at t distance between $\mathcal{T}(P_0)$ and $\mathcal{T}(P_1)$ on the transformed line.

Note that while ratios of distances remain constant, angles and exact distances don't necessarily stay the same. The specific subset of affine transformations that preserve these features are called *rigid transformations*; those that don't are called *deformations*. It should be no surprise that we find rigid transformations useful. When transforming our models, in most cases we don't want them distorted unrecognizably. A bottle should maintain its size and shape—it should look like a bottle no matter where we place it in space. However, the deformations have their use as well. On occasion we may want to make an object larger or smaller or reflect it across a plane, as in a mirror.

To apply an affine transformation to a vector in an affine space, we can apply it to the difference of two points that equal the vector, or

$$\mathcal{T}(\mathbf{v}) = \mathcal{T}(P - Q) = \mathcal{T}(P) - \mathcal{T}(Q)$$

So, as we've seen above, an affine transformation that is applied to a vector performs a linear transformation.

4.2.3 Formal Representation

Suppose we have an affine transformation that maps from affine space A to affine space B, where the frame for A has basis vectors $(\mathbf{v}_0, \ldots, \mathbf{v}_{n-1})$ and origin O_A, and the frame for B has basis vectors $(\mathbf{w}_0, \ldots, \mathbf{w}_{m-1})$ and origin O_B. If we apply an affine transformation to a point $P = (x_0, \ldots, x_{n-1})$ in A, this gives

$$\begin{aligned}\mathcal{T}(P) &= \mathcal{T}(x_0\mathbf{v}_0 + \cdots + x_{n-1}\mathbf{v}_{n-1} + O_A) \\ &= x_0\mathcal{T}(\mathbf{v}_0) + \cdots + x_{n-1}\mathcal{T}(\mathbf{v}_{n-1}) + \mathcal{T}(O_A)\end{aligned}$$

As we did with linear transformations, we can express a given $\mathcal{T}(\mathbf{v})$ in terms of B's frame:

$$\mathcal{T}(\mathbf{v}_j) = a_{0,j}\mathbf{w}_0 + a_{1,j}\mathbf{w}_1 + \cdots + a_{m-1,j}\mathbf{w}_{m-1}$$

Similarly, we can express $\mathcal{T}(O_A)$ in terms of B's frame:

$$\mathcal{T}(O_A) = y_0\mathbf{w}_0 + y_1\mathbf{w}_1 + \cdots + y_{m-1}\mathbf{w}_{m-1} + O_B$$

Again, as we did with linear transformations, we can rewrite this as a matrix product. However, unlike linear transformations, we write a mapping from an n-dimensional affine space to an m-dimensional affine space as an $(m+1) \times (n+1)$ matrix:

$$\begin{bmatrix} a_{0,0}\mathbf{w}_0 & a_{0,1}\mathbf{w}_0 & \cdots & a_{0,n-1}\mathbf{w}_0 & y_0\mathbf{w}_0 \\ a_{1,0}\mathbf{w}_1 & a_{1,1}\mathbf{w}_1 & \cdots & a_{1,n-1}\mathbf{w}_1 & y_1\mathbf{w}_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0}\mathbf{w}_{m-1} & a_{m-1,1}\mathbf{w}_{m-1} & \cdots & a_{m-1,n-1}\mathbf{w}_{m-1} & y_{m-1}\mathbf{w}_{m-1} \\ 0 & 0 & \cdots & 0 & O_B \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ 1 \end{bmatrix}$$

The dimensions of our matrix now make sense. The $n+1$ columns represent the n transformed basis vectors plus the transformed origin. We need $m+1$ rows since the frame of B has m basis vectors plus the origin O_B.

We can pull out the frame terms to get

$$\begin{bmatrix}\mathbf{w}_0 & \mathbf{w}_1 & \cdots & \mathbf{w}_{m-1} & O_B\end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} & y_0 \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} & y_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} & y_{m-1} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ 1 \end{bmatrix}$$

So, similar to linear transformations, if we know how the affine transformation affects the frame for A, we can copy the transformed frame *in terms*

of the frame for B into the columns of a matrix and use matrix multiplication to apply the affine transformation to an arbitrary point.

4.3 Standard Affine Transformations

Now that we've defined affine transformations in general, we can discuss some specific affine transformations that will prove useful when manipulating objects in our game. We'll cover these in terms of transformations from $\mathbb{R}^3$ to $\mathbb{R}^3$, since they will be the most common uses. However, we can apply similar principles to find transformations from $\mathbb{R}^2$ to $\mathbb{R}^2$ or even $\mathbb{R}^4$ to $\mathbb{R}^4$ if we desire.

Since affine spaces A and B are the same in this case, to simplify things we'll use the same frame for each one: the standard Cartesian frame of $(\mathbf{i}, \mathbf{j}, \mathbf{k}, O)$.

4.3.1 Translation

The most basic affine transformation is *translation*. For a single point, it's the same as adding a vector $\mathbf{t}$ to it, and when applied to an entire set of points it has the effect of moving them rigidly through space (Figure 4.1). Since all the points are shifted equally in space, the size and shape of the object will not change, so this is a rigid transformation.

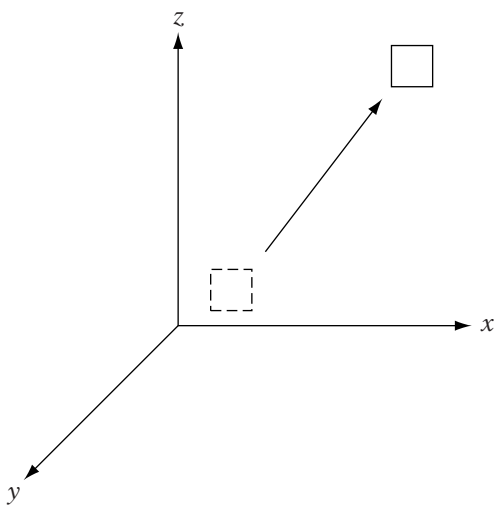


Figure 4.1 Translation.

We can determine the matrix for a translation by computing the transformation for each of the frame elements. For the origin O, this is

$$\begin{aligned}\mathcal{T}(O) &= \mathbf{t} + O \\ &= t_x\mathbf{i} + t_y\mathbf{j} + t_z\mathbf{k} + O\end{aligned}$$

For a given basis vector, we can find two points P and Q that define the vector and compute the transformation of their difference. For example, for $\mathbf{i}$:

$$\begin{aligned}\mathcal{T}(\mathbf{i}) &= \mathcal{T}(P - Q) \\ &= \mathcal{T}(P) - \mathcal{T}(Q) \\ &= (\mathbf{t} + P) - (\mathbf{t} + Q) \\ &= P - Q \\ &= \mathbf{i}\end{aligned}$$

The same holds true for $\mathbf{j}$ and $\mathbf{k}$, so translation has no effect on the basis vectors in our frame. We end up with a 4×4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Or, in block form:

$$\mathbf{T}_\mathbf{t} = \begin{bmatrix} \mathbf{I} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Translation only affects points. To see why, suppose we have a vector $\mathbf{v}$, which equals the displacement between two points P and Q; that is, $\mathbf{v} = P - Q$. If we translate $P - Q$, we get

$$\begin{aligned}\text{trans}(P - Q) &= (P + \mathbf{t}) - (Q + \mathbf{t}) \\ &= (P - Q) + (\mathbf{t} - \mathbf{t}) \\ &= \mathbf{v}\end{aligned}$$

This fits with our geometric notion that points have position and hence can be translated in space, while vectors do not and cannot.

We can use equation 4.3 to compute the inverse translation transformation:

$$\mathbf{T}_{\mathbf{t}}^{-1} = \begin{bmatrix} \mathbf{I}^{-1} & -\mathbf{I}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.4}$$

$$= \begin{bmatrix} \mathbf{I} & -\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{4.5}$$

$$= \mathbf{T}_{-\mathbf{t}} \tag{4.6}$$

So, the inverse of a given translation negates the original translation vector to displace the point back to its original position.

4.3.2 Rotation

The other common rigid transformation is *rotation*. If we consider the rotation of a vector, we are rigidly changing its direction around an axis without changing its length. In $\mathbb{R}^2$, this is the same as replacing a vector with the one that's θ degrees counterclockwise (Figure 4.2).

In $\mathbb{R}^3$, we usually talk about an *axis of rotation*. In his rotation theorem, Euler showed that when applying a rotation in three-dimensional (3D) space, there is a linear set of points (i.e., a line) that does not change. This is called the axis of rotation, and the amount we rotate around this axis is the *angle of rotation*. A helpful mnemonic is the right-hand rule: If you point your right thumb in the direction of the axis vector, the curl of your fingers represents the direction of positive rotation (Figure 4.3).

For a given point, we rotate it by moving it along a planar arc a constant distance from another point, known as the *center of rotation* (Figure 4.4). This

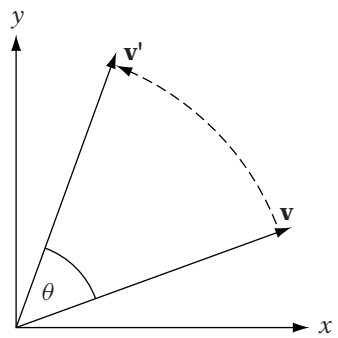


Figure 4.2 Rotation of vector in R^2.

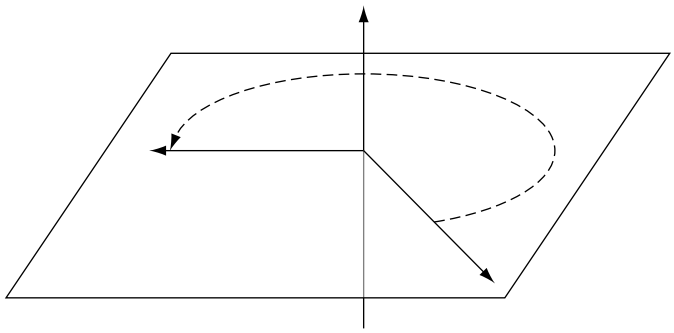


FIGURE 4.3 Axis and plane of rotation.

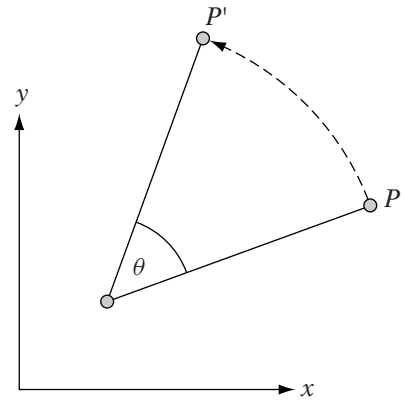


FIGURE 4.4 Rotation of point in R^2.

center of rotation is commonly defined as the origin of the current frame (we'll refer to this as a *pure rotation*) but can be any arbitrary point. We can think of this as defining a vector **v** from the center of rotation to the point to be rotated, rotating **v**, and then adding the result to the center of rotation to compute the new position of the point. For now we'll only cover pure rotations; applying general affine transformations about an arbitrary center will be discussed later.

To keep things simple, we'll begin with rotations around one of the three frame axes, with a center of rotation equal to the origin. The following system of equations rotates a vector or point counterclockwise (assuming the axis is pointing at us) around **k**, or the z-axis (Figure 4.5c):

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \\ z' &= z \end{aligned} \tag{4.7}$$

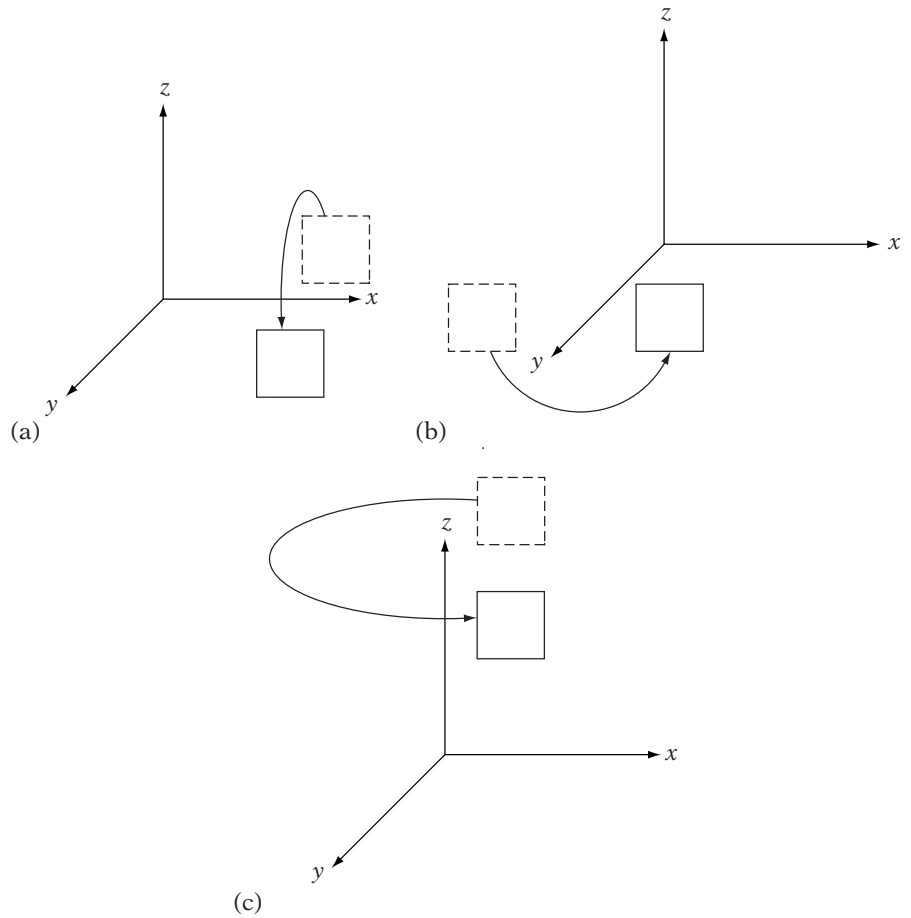


FIGURE 4.5 (a) x-axis rotation, (b) y-axis rotation, and (c) z-axis rotation.

Figure 4.6 shows why this works. Since we're rotating around the z-axis, no z values will change, so we will consider only how the rotation affects the xy values of the points. The starting position of the point is (x, y), and we want to rotate that θ degrees counterclockwise. Handling this in Cartesian coordinates can be problematic, but this is one case where polar coordinates are useful.

Recall that a point P in polar coordinates has representation (r, ϕ), where r is the distance from the origin and ϕ[1] is the counterclockwise angle from the x-axis. We can think of this as rotating an r length radius lying along the x-axis by ϕ degrees. If we rotate this a further θ degrees, the end of the radius

1. We're using ϕ for polar coordinates in this case to distinguish it from the rotation angle θ.

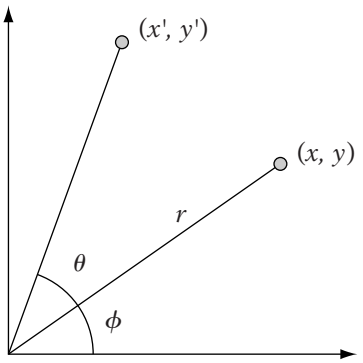


FIGURE 4.6 Rotation in xy plane.

will be at $(r, \phi + \theta)$ (in polar coordinates). Converting to Cartesian coordinates, the final point will lie at

$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

Using trigonometric identities, this becomes

$$x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$

But $r\cos\phi = x$, and $r\sin\phi = y$, so we can substitute and get

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

We can derive similar equations for rotation around the x-axis (Figure 4.5a):

$$x' = x$$
$$y' = y\cos\theta - z\sin\theta$$
$$z' = y\sin\theta + z\cos\theta$$

and rotation around the y-axis (Figure 4.5b):

$$x' = z\sin\theta + x\cos\theta$$
$$y' = y$$
$$z' = z\cos\theta - x\sin\theta$$

To create the corresponding transformation, we need to determine how the frame elements are transformed. The frame's origin will not change since it's our center of rotation, so $\mathbf{y} = \mathbf{0}$. Therefore, our primary concern will be the contents of the 3×3 matrix $\mathbf{A}$.

For this matrix, we need to compute where $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ will go. For example, for rotations around the z-axis we can transform $\mathbf{i}$ to get

$$\begin{aligned} x' &= (1)\cos\theta - (0)\sin\theta = \cos\theta \\ y' &= (1)\sin\theta + (0)\cos\theta = \sin\theta \\ z' &= 0 \end{aligned}$$

Transforming $\mathbf{j}$ and $\mathbf{k}$ similarly and copying the results into the columns of a 3×3 matrix gives

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Similar matrices can be created for rotation around the x-axis:

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

and around the y-axis:

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

One thing to note about these matrices is that their determinants are equal to 1, and they are all orthogonal. For example, look at the component 3-vectors of the z-axis rotation matrix. We have $(\cos\theta, \sin\theta, 0)$, $(-\sin\theta, \cos\theta, 0)$, and $(0, 0, 1)$. The first two lie on the xy plane and so are perpendicular to the third, and they are perpendicular to each other. All three are unit length and so form an orthonormal basis.

The product of two orthogonal matrices is also an orthogonal matrix, thus the product of a series of pure rotation matrices is also a rotation matrix. For example, by concatenating matrices that rotate around the z-axis, then the

y-axis, and then the x-axis, we can create one form of a generalized rotation matrix:

$$\mathbf{R}_x\mathbf{R}_y\mathbf{R}_z = \begin{bmatrix} CyCz & -CySz & Sy \\ SxSyCz + CxSz & -SxSySz + CxCz & -SxCy \\ -CxSyCz + SxSz & CxSySz + SxCz & CxCy \end{bmatrix} \tag{4.8}$$

where

$$\begin{aligned} Cx &= \cos\theta_x \quad Sx = \sin\theta_x \\ Cy &= \cos\theta_y \quad Sy = \sin\theta_y \\ Cz &= \cos\theta_z \quad Sz = \sin\theta_z \end{aligned}$$

Recall that the inverse of an orthogonal matrix is its transpose. Because pure rotation matrices are orthogonal, the inverse of any rotation matrix is also its transpose. Therefore, the inverse of the z-axis rotation, centered on the origin, is

$$\mathbf{R}_z^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

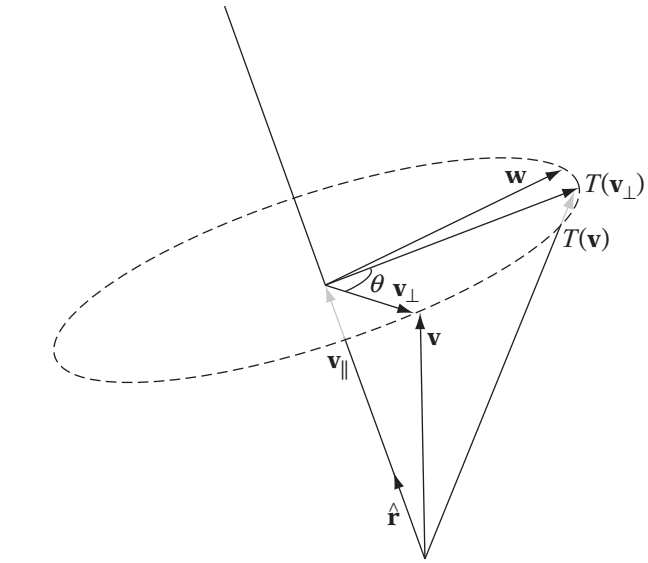
This follows if we think of the inverse transformation as "undoing" the original transformation. If you substitute $-\theta$ for θ in the original matrix and replace $\cos(-\theta)$ with $\cos\theta$ and $\sin(-\theta)$ with $-\sin\theta$, then we have:

$$\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

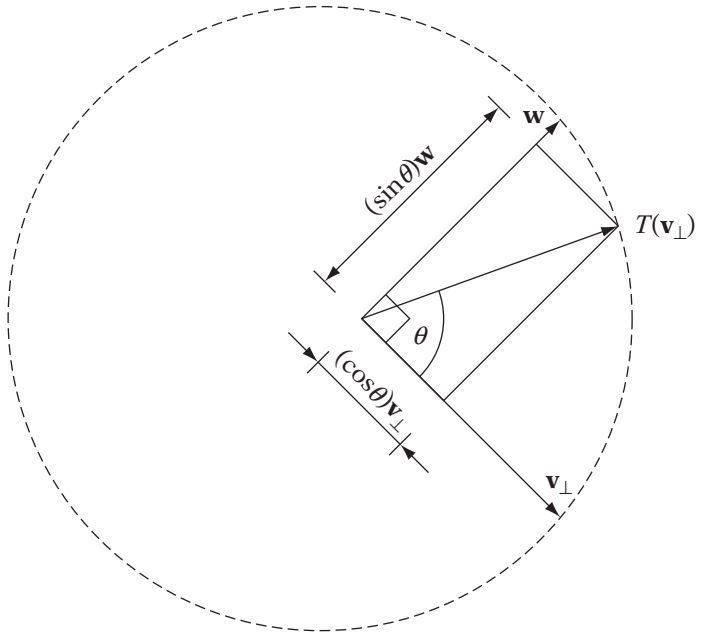
which, as we can see, results in the immediately preceding inverse matrix.

Now that we have looked at rotations around the coordinate axes, we will consider rotations around an arbitrary axis. The formula for a rotation of a vector $\mathbf{v}$ by an angle θ around a general axis $\hat{\mathbf{r}}$ is derived as follows. We begin by breaking $\mathbf{v}$ into two parts: the part parallel with $\hat{\mathbf{r}}$ and the part perpendicular to it, which lies on the plane of rotation (Figure 4.7a). Recall from Chapter 1 that the parallel part $\mathbf{v}_\parallel$ is the projection of $\mathbf{v}$ onto $\hat{\mathbf{r}}$, or

$$\mathbf{v}_\parallel = (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} \tag{4.9}$$



(a)



(b)

FIGURE 4.7 (a) General rotation, showing axis of rotation and rotation plane, and (b) general rotation, showing vectors on rotation plane.

The perpendicular part is what remains of $\mathbf{v}$ after we subtract the parallel part, or

$$\mathbf{v}_\perp = \mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} \tag{4.10}$$

To properly compute the effect of rotation, we need to create a two-dimensional (2D) basis on the plane of rotation (Figure 4.7b). We'll use $\mathbf{v}_\perp$ as our first basis vector, and we'll need a vector $\mathbf{w}$ perpendicular to it for our second basis vector. We can take the cross product with $\hat{\mathbf{r}}$ for this:

$$\mathbf{w} = \hat{\mathbf{r}} \times \mathbf{v}_\perp = \hat{\mathbf{r}} \times \mathbf{v} \tag{4.11}$$

In the standard basis for $\mathbb{R}^2$, if we rotate the vector $\mathbf{i} = (1, 0)$ by θ, we get the vector $(\cos\theta, \sin\theta)$. Equivalently,

$$R\mathbf{i} = (\cos\theta)\mathbf{i} + (\sin\theta)\mathbf{j}$$

If we use $\mathbf{v}_\perp$ and $\mathbf{w}$ as the 2D basis for the rotation plane, we can find the rotation of $\mathbf{v}_\perp$ by θ in a similar manner:

$$R\mathbf{v}_\perp = (\cos\theta)\mathbf{v}_\perp + (\sin\theta)\mathbf{w} \tag{4.12}$$

The parallel part of $\mathbf{v}$ doesn't change with the rotation, so the final result of rotating $\mathbf{v}$ around $\hat{\mathbf{r}}$ by θ is

$$\begin{aligned} R\mathbf{v} &= R\mathbf{v}_\parallel + R\mathbf{v}_\perp \\ &= R\mathbf{v}_\parallel + (\cos\theta)\mathbf{v}_\perp + (\sin\theta)\mathbf{w} \\ &= (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} + \cos\theta[\mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}] + \sin\theta(\hat{\mathbf{r}} \times \mathbf{v}) \\ &= \cos\theta\mathbf{v} + [1 - \cos\theta](\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}} \times \mathbf{v}) \end{aligned} \tag{4.13}$$

This is one form of what is known as the *Rodrigues formula*.

The projection $(\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}}$ can be replaced by the tensor product $(\hat{\mathbf{r}} \otimes \hat{\mathbf{r}})\mathbf{v}$. Similarly, the cross product $\hat{\mathbf{r}} \times \mathbf{v}$ can be replaced by a multiplication by a skew symmetric matrix $\tilde{\mathbf{r}}\mathbf{v}$. This gives

$$\begin{aligned} R\mathbf{v} &= \cos\theta\mathbf{v} + (1 - \cos\theta)(\hat{\mathbf{r}} \otimes \hat{\mathbf{r}})\mathbf{v} + \sin\theta\tilde{\mathbf{r}}\mathbf{v} \\ &= [\cos\theta\mathbf{I} + (1 - \cos\theta)(\hat{\mathbf{r}} \otimes \hat{\mathbf{r}}) + \sin\theta\tilde{\mathbf{r}}]\mathbf{v} \end{aligned}$$

Expanding the terms, we end up with a matrix:

$$\mathbf{R}_{\hat{\mathbf{r}}\theta} = \begin{bmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + c \end{bmatrix}$$

where

$$\hat{\mathbf{r}} = (x, y, z)$$
$$c = \cos\theta$$
$$s = \sin\theta$$
$$t = 1 - \cos\theta$$

As we can see, there is a wide variety of choices for the 3×3 matrix $\mathbf{A}$, depending on what sort of rotation we wish to perform. The full affine matrix for rotation around the origin is

$$\begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\mathbf{R}$ is one of the rotation matrices just given. For example, the affine matrix for rotation around the x-axis is

$$\begin{bmatrix} \mathbf{R}_x & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is also an orthogonal matrix and its inverse is the transpose, as before.

Finally, when discussing rotations one has to be careful to distinguish rotation from orientation, which is to rotation as position is to translation. If we consider the representation of a point in an affine space,

$$P = \mathbf{v} + O$$

then we can think of the origin as a reference position and the vector $\mathbf{v}$ as a translation that relates our position to the reference. We can represent our position as just the components of the translation. Similarly, we can define a reference orientation Ω_0, and any orientation Ω is related to it by a rotation, or

$$\Omega = \mathbf{R}_0\Omega_0$$

Just as we might use the components of the vector $\mathbf{v}$ to represent our position, we can use the rotation $\mathbf{R}_0$ to represent our orientation. To change our orientation, we apply an additional rotation, just as we might add a translation vector to change our position:

$$\Omega' = \mathbf{R}_1\Omega$$

In this case, our final orientation, using the rotation component, is

$$\mathbf{R}_1\mathbf{R}_0$$

Remember that the order of concatenation matters, because matrix multiplication — particularly for rotation matrices — is not a commutative operation.

4.3.3 Scaling

The remaining affine transformations that we will cover are *deformations*, since they don't preserve exact lengths or angles. The first is *scaling*, which can be thought of as corresponding to our other basic vector operation, scalar multiplication; however, it is not quite the same. Scalar multiplication of a vector has only one multiplicative factor and changes a vector's length equally in all directions. We can also multiply a vector by a negative scalar. In comparison, scaling as it is commonly used in computer graphics applies a possibly different but positive factor to each basis vector in our frame.[2] If all the factors are equal, then it is called uniform scaling and is — for vectors in the affine space — equivalent to scalar multiplication by a single positive scalar. Otherwise, it is called nonuniform scaling. Full nonuniform scaling can be applied differently in each axis direction, so we can scale by 2 in z to make an object twice as tall, but 1/2 in x and y to make it half as wide.

A point doesn't have a length per se, so instead we change its relative distance from another point C_s, known as the center of scaling. We can consider this as scaling the vector from the center of scaling to our point P. For a set of points, this will end up scaling their distance relative to each other, but still maintaining the same relative shape (Figure 4.8).

For now we'll consider only scaling around the origin, so $C_s = O$ and $\mathbf{y} = \mathbf{0}$. For the upper 3×3 matrix $\mathbf{A}$, we again need to determine how the frame basis vectors change, which is defined as

$$\mathcal{T}(\mathbf{i}) = a\mathbf{i}$$

$$\mathcal{T}(\mathbf{j}) = b\mathbf{j}$$

$$\mathcal{T}(\mathbf{k}) = c\mathbf{k}$$

where $a, b, c > 0$ and are the scale factors in the x, y, z directions, respectively. Writing these transformed basis vectors as the columns of $\mathbf{A}$, we get an affine matrix of

$$\mathbf{S}_{abc} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. We'll consider negative factors when we discuss reflections in the following section.

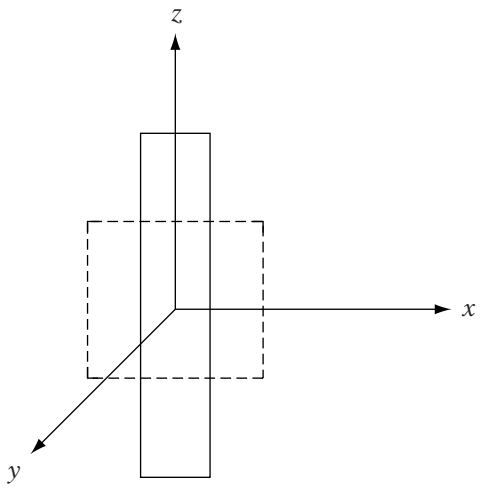


FIGURE 4.8 Nonuniform scaling.

This is a diagonal matrix, with the positive scale factors lying along the diagonal, so the inverse is

$$\mathbf{S}_{abc}^{-1} = \mathbf{S}_{\frac{1}{a}\frac{1}{b}\frac{1}{c}} = \begin{bmatrix} 1/a & 0 & 0 & 0 \\ 0 & 1/b & 0 & 0 \\ 0 & 0 & 1/c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4.3.4 REFLECTION

The reflection transformation symmetrically maps an object across a plane or through a point. One possible reflection is (Figure 4.9a)

$$\begin{aligned} x' &= -x \\ y' &= y \\ z' &= z \end{aligned}$$

This reflects across the yz plane and gives an effect like a standard mirror (mirrors don't swap left to right, they swap front to back). If we want to reflect across the xz plane instead, we would use (Figure 4.9b)

$$\begin{aligned} x' &= x \\ y' &= -y \\ z' &= z \end{aligned}$$

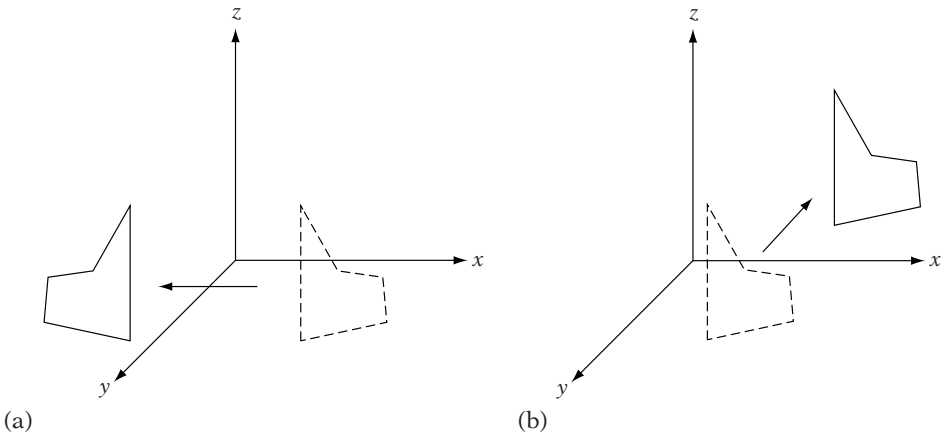


FIGURE 4.9 (a) *yz* reflection, and (b) *xz* reflection.

As one might expect, we can create a planar reflection that reflects across a general plane, defined by a normal $\hat{\mathbf{n}}$ and a point on the plane P_0. For now we'll consider only planes that pass through the origin. If we have a vector $\mathbf{v}$ in our affine space, we can break it into two parts relative to the plane normal: the orthogonal part $\mathbf{v}_\perp$, which will remain unchanged, and parallel part $\mathbf{v}_\parallel$, which will be reflected to the other side of the plane to become $-\mathbf{v}_\parallel$. The transformed vector will be the sum of $\mathbf{v}_\perp$ and the reflected $-\mathbf{v}_\parallel$ (Figure 4.10).

To compute $\mathbf{v}_\parallel$, we merely have to take the projection of $\mathbf{v}$ against the plane normal $\hat{\mathbf{n}}$, or

$$\mathbf{v}_\parallel = (\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{4.14}$$

Subtracting this from $\mathbf{v}$, we can compute $\mathbf{v}_\perp$:

$$\mathbf{v}_\perp = \mathbf{v} - \mathbf{v}_\parallel \tag{4.15}$$

We know that the transformed vector will be $\mathbf{v}_\perp - \mathbf{v}_\parallel$. Substituting equations 4.15 and 4.14 into this gives us

$$\begin{aligned}\mathcal{T}(\mathbf{v}) &= \mathbf{v}_\perp - \mathbf{v}_\parallel \\ &= \mathbf{v} - 2\mathbf{v}_\parallel \\ &= \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}\end{aligned}$$

From Chapter 2, we know that we can perform the projection of $\mathbf{v}$ on $\hat{\mathbf{n}}$ by multiplying by the tensor product matrix $\hat{\mathbf{n}} \otimes \hat{\mathbf{n}}$, so this becomes

$$\begin{aligned}\mathcal{T}(\mathbf{v}) &= \mathbf{v} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})\mathbf{v} \\ &= [\mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})]\mathbf{v}\end{aligned}$$

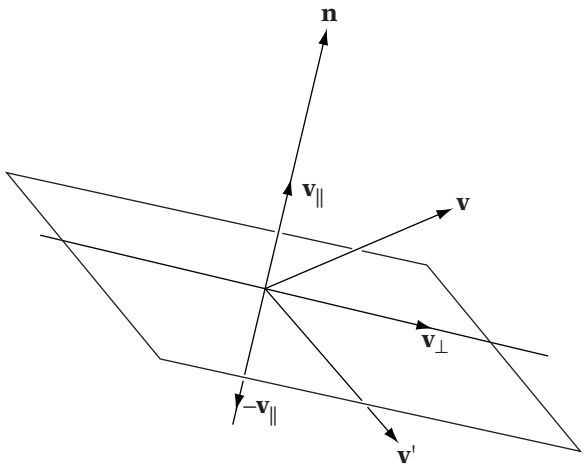


FIGURE 4.10 General reflection.

Thus, the linear transformation part **A** of our affine transformation is $[\mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}})]$. Writing this as a block matrix, we get:

$$\mathbf{F}_{\mathbf{n}} = \begin{bmatrix} \mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}}) & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

While in the real world we usually see planar reflections, in our virtual world we can also compute a reflection through a point. The following performs a reflection through the origin (Figure 4.11):

$$\begin{aligned} x' &= -x \\ y' &= -y \\ z' &= -z \end{aligned}$$

The corresponding block matrix is

$$\mathbf{F}_O = \begin{bmatrix} -\mathbf{I} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Reflections are a symmetric operation; that is, the reflection of a reflection returns the original point or vector. Because of this, the inverse of a reflection matrix is the matrix itself.

As an aside, we would (incorrectly) expect that if we can reflect through a plane and a point, we can reflect through a line. The system

$$\begin{aligned} x' &= -x \\ y' &= -y \\ z' &= z \end{aligned}$$

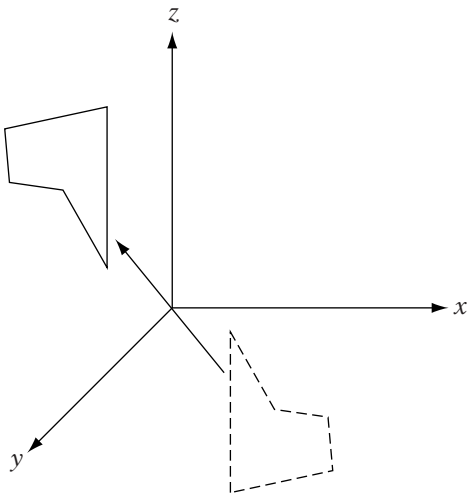


FIGURE 4.11 Point reflection.

appears to reflect through the z-axis, giving a "funhouse mirror" effect, where right and left are swapped (if y is left, it becomes $-y$ in the reflection, and so ends up on the right side). However, if we examine the transformation closely, we see that while it does perform the desired effect, this is actually a rotation of 180 degrees around the z-axis. While both pure rotations and pure reflections through the origin are orthogonal matrices, we can distinguish between them by noting that reflection matrices have a determinant of -1, while rotation matrices have a determinant of 1.

4.3.5 SHEAR

The final affine transformation that we will cover is shear. Because it affects the angles of objects it is not used all that often, but it comes up particularly when discussing oblique projections. An axis-aligned shear provides a shift in one or two axes proportional to the component in a third axis. Transforming a square to a rhombus or a cube to a rhomboid solid is a shear transformation (Figure 4.12).

There are a number of ways of specifying shear [82, 100]. In our case, we will define a shear plane, with normal $\hat{\mathbf{n}}$, that does not change due to the transformation. We define an orthogonal shear vector $\mathbf{s}$, which indicates how planes parallel to the shear plane will be transformed. Points on the plane 1 unit of distance from the shear plane, in the direction of the plane normal, will be displaced by $\mathbf{s}$. Points on the plane 2 unit of distance from the shear plane will be displaced by $2\mathbf{s}$, and so on. In general, if we take a point P and define

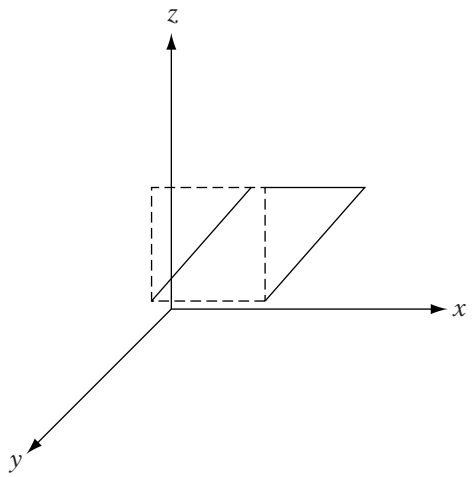


FIGURE 4.12 z-shear on square.

it as $P_0 + \mathbf{v}$, where P_0 is a point on the shear plane, then P will be displaced by $(\hat{\mathbf{n}} \cdot \mathbf{v})\mathbf{s}$.

The simplest case is when we apply shear perpendicular to one of the main coordinate axes. For example, if we take the yz plane as our shear plane, our normal is $\mathbf{i}$ and the shear plane passes through the origin O. We know from this that O will not change with the transformation, so our translation vector $\mathbf{y}$ is $\mathbf{0}$. As before, to find $\mathbf{A}$ we need to figure out how the transformation affects our basis vectors. If we define $\mathbf{j}$ as $P_1 - O$, then

$$\mathcal{T}(\mathbf{j}) = \mathcal{T}(P_1) - \mathcal{T}(O)$$

But P_1 and O lie on the shear plane, so

$$\begin{aligned}\mathcal{T}(\mathbf{j}) &= P_1 - O \\ &= \mathbf{j}\end{aligned}$$

The same is true for the basis vector $\mathbf{k}$. For $\mathbf{i}$, we can define it as $P_0 - O$. We know that P_0 is distance 1 from the shear plane, so it will become $P_0 + \mathbf{s}$, so

$$\begin{aligned}\mathcal{T}(\mathbf{i}) &= \mathcal{T}(P_0) - \mathcal{T}(O) \\ &= P_0 + \mathbf{s} - O \\ &= \mathbf{i} + \mathbf{s}\end{aligned}$$

The vector $\mathbf{s}$ in this case is orthogonal to $\mathbf{i}$, therefore it is of the form $(0, a, b)$, so our transformed basis vector will be $(1, a, b)$. Our final matrix $\mathbf{A}$ is

$$\mathbf{H}_x = \begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & 0 & 1 \end{bmatrix}$$

We can go through a similar process to get shear by the y-axis:

$$\mathbf{H}_y = \begin{bmatrix} 1 & c & 0 \\ 0 & 1 & 0 \\ 0 & d & 1 \end{bmatrix}$$

and shear by the z-axis:

$$\mathbf{H}_z = \begin{bmatrix} 1 & 0 & e \\ 0 & 1 & f \\ 0 & 0 & 1 \end{bmatrix}$$

For shearing by a general plane through the origin, we already have the formula for the displacement: $(\hat{\mathbf{n}} \cdot \mathbf{v})\mathbf{s}$. We can rewrite this as a tensor product to get $(\hat{\mathbf{n}} \otimes \mathbf{s})\mathbf{v}$. Because this is merely the displacement, we need to include the original point, and thus our origin-centered general shear matrix is simply $\mathbf{I} + \hat{\mathbf{n}} \otimes \mathbf{s}$. Our final shear matrix is

$$\mathbf{H}_{\hat{\mathbf{n}},\mathbf{s}} = \begin{bmatrix} \mathbf{I} + \mathbf{s} \otimes \hat{\mathbf{n}} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

The inverse shear transformation is shear in the opposite direction, so the corresponding matrix is

$$\mathbf{H}^{-1}_{\hat{\mathbf{n}},\mathbf{s}} = \begin{bmatrix} \mathbf{I} - \mathbf{s} \otimes \hat{\mathbf{n}} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} = \mathbf{H}_{\hat{\mathbf{n}},-\mathbf{s}}$$

4.3.6 Applying an Affine Transformation Around an Arbitrary Point

Up to this point, we have been assuming that our affine transformations are applied around the origin of the frame. For example, when discussing rotation we treated the origin as our center of rotation. Similarly, our shear planes were assumed to pass through the origin. This doesn't necessarily have to be the case.

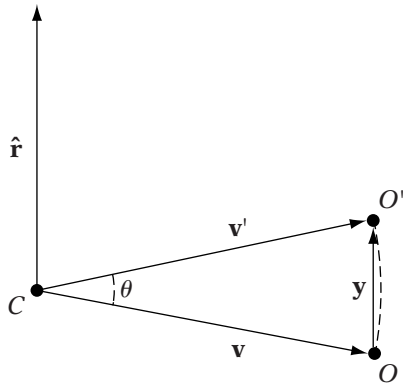


FIGURE 4.13 Rotation of origin around arbitrary center.

Let's look at a particular example—the rotation of a point around an arbitrary center of rotation C—and determine how this transformation affects the origin of our frame. If we look at Figure 4.13, we see the situation. We have a point C and our origin O. We want to rotate the difference vector $\mathbf{v} = O - C$ between the two points by matrix $\mathbf{R}$ and determine where the resulting point $\mathcal{T}(O)$, or $C + \mathcal{T}(\mathbf{v})$, will be. From that we can compute the difference vector $\mathbf{y} = \mathcal{T}(O) - O$. From Figure 4.13, we can see that $\mathbf{y} = \mathcal{T}(\mathbf{v}) - \mathbf{v}$, so we can reduce this as follows:

$$\begin{aligned}\mathbf{y} &= \mathcal{T}(\mathbf{v}) - \mathbf{v} \\ &= \mathbf{R}\mathbf{v} - \mathbf{v} \\ &= (\mathbf{R} - \mathbf{I})\,\mathbf{v}\end{aligned}$$

It's usually more convenient to write this in terms of the vector dual to C, which is $\mathbf{x} = C - O = -\mathbf{v}$, so this becomes

$$\begin{aligned}\mathbf{y} &= -(\mathbf{R} - \mathbf{I})\,\mathbf{x} \\ &= (\mathbf{I} - \mathbf{R})\,\mathbf{x}\end{aligned}$$

We can achieve the same result by translating our center C to the frame origin by $-\mathbf{x}$, performing our origin-centered rotation, and then translating back by $\mathbf{x}$:

$$\begin{aligned}\mathbf{M}_c &= \begin{bmatrix} \mathbf{I} & \mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & -\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{R} & \mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} \mathbf{I} & -\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{R} & (\mathbf{I} - \mathbf{R})\,\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}\end{aligned}$$

Notice that the upper left-hand block $\mathbf{R}$ is not affected by this process.

The same construction can be used for all affine transformations that use a center of transformation: rotation, scale, reflection, and shear. The exception is translation, since such an operation has no effect: $P - \mathbf{x} + \mathbf{t} + \mathbf{x} = P + \mathbf{t}$. But for the others, using a point $C = (\mathbf{x}, 1)$ as our arbitrary center of transformation gives

$$\mathbf{M}_c = \begin{bmatrix} \mathbf{A} & (\mathbf{I} - \mathbf{A})\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where $\mathbf{A}$ is the upper 3×3 matrix of an origin-centered transformation. The corresponding inverse is

$$\mathbf{M}_c^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & (\mathbf{I} - \mathbf{A}^{-1})\mathbf{x} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

4.3.7 Transforming Plane Normals

As we saw in the previous section, if we want to transform a line or plane represented in parametric form, we transform the points in the affine combination. For example,

$$\mathcal{T}(P(t)) = (1 - s - t)\mathcal{T}(P_0) + s\mathcal{T}(P_1) + t\mathcal{T}(P_2)$$

But suppose we have a plane represented using the generalized plane equation. One way of considering this is as a plane normal (a, b, c) and a point on the plane P_0. We could transform these and try to use the resulting vector and point to build the new plane. However, if we apply an affine transform to the plane normal (a, b, c) directly, we may end up performing a deformation. Since angles aren't preserved under deformations, the resulting normal may no longer be orthogonal to the points in the plane.

The correct approach is as follows. We can represent the generalized plane equation as the product of a row matrix and column matrix, or

$$\begin{aligned} ax + by + cz + d &= \begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\ &= \mathbf{n}^T P \end{aligned}$$

Now P is clearly a point, and $\mathbf{n}$ is the vector of coefficients for the plane. For points that lie on the plane,

$$\mathbf{n}^T P = 0$$

If we transform all the points on the plane by some matrix $\mathbf{M}$, then to maintain the relationship between $\mathbf{n}^T$ and P, we'll have to transform $\mathbf{n}$ by some unknown matrix $\mathbf{Q}$, or

$$(\mathbf{Q}\mathbf{n})^T(\mathbf{M}P) = 0$$

This can be rewritten as

$$\mathbf{n}^T\mathbf{Q}^T\mathbf{M}P = 0$$

One possible solution for this is if

$$\mathbf{I} = \mathbf{Q}^T\mathbf{M}$$

Solving for $\mathbf{Q}$ gives

$$\mathbf{Q} = \left(\mathbf{M}^{-1}\right)^T$$

So, the transformed plane coefficients become

$$\mathbf{n}' = \left(\mathbf{M}^{-1}\right)^T \mathbf{n}$$

The same approach will work if we're transforming the plane normal and point as described earlier. We transform the point P_0 by $\mathbf{M}$ and the normal by $(\mathbf{M}^{-1})^T$.

In many cases the inverse matrix $\mathbf{M}^{-1}$ may not exist. So, if we're just transforming a normal vector (a, b, c), we can use a different method. Instead of $\mathbf{M}^{-1}$, we use the adjoint matrix from Cramer's rule. Normally we couldn't proceed at this point: If the inverse doesn't exist, we end up dividing by a zero determinant. However, even when the inverse exists, the division by the determinant is a scale factor. So, we can ignore it in all cases and just use the adjoint matrix directly, because we're going to normalize the resulting vector anyway.

4.4 Using Affine Transformations

4.4.1 Manipulation of Game Objects

The primary use of affine transformations is for the manipulation of objects in our game world. Suppose, from our earlier hypothetical, we have an office environment that is acting as our game space. The artists could build

the basic level — the walls, the floor, the ceilings, and so forth — as a single set of triangles with coordinates defined to place them exactly where we might want them in the world. However, suppose we have a single desk model that we want to duplicate and place in various locations in the level. The artist could build a new version of the desk for each location in the core level geometry, but that would involve unnecessarily duplicating all the memory needed for the model. Instead, we could have one version, or *master*, of the desk model and then set a series of transformations that indicate where in the level each copy, or *instance*, of the desk should be placed [108].

Before we can begin to discuss how we specify these transformations and what they might mean, we need to define the two different coordinate frames we are working in: the local coordinate frame and the world coordinate frame.

Local and World Coordinate Frames

When artists create an object or we create an object directly in a program, the coordinates of the points that make up that object are defined in that particular object's *local frame*. This is also commonly known as *local space*. In addition, often the frame is named after the object itself, so you might also see terms like *model space* or *camera space*.

The orientation of the basis vectors in the lcoal frame is usually set so that the engineers know which part of the object is the front, which is the top, and which is the side. This allows us to orient the object correctly relative to the rest of the world and to translate it in the correct direction if we want to move it forward. The convention that we will be using in this book is one where the x-axis points along the forward direction of the object, the y-axis points toward the left of the object, and the z-axis points out the top of the object (Figure 4.14). Another common convention is to use the y-axis for up, the z-axis for forward, and the x-axis for either out to the left or to the right, depending on whether we want to work in a right-handed or left-handed frame.

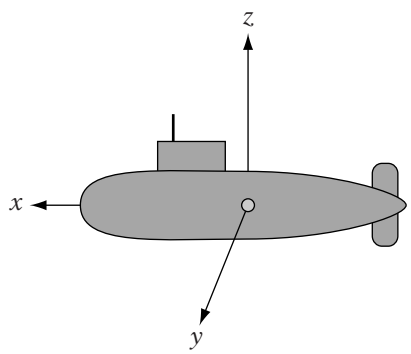


Figure 4.14 Local object frame.

Typically, the origin of the frame is placed in a position convenient for the game, either at the center of the object or at the bottom of the object. The first is useful when we want to rotate objects around their centers, the second for placement on the ground.

When constructing our world, we define a specific coordinate frame, or *world frame*, also known as *world space*. The world frame acts as a common reference among all the objects, much as the origin acts as a common reference among points. Ultimately, in order to render, simulate, or otherwise interact with objects, we will need to transform their local coordinates into the world frame.

When an artist builds the level geometry, the coordinates are usually set in the world frame. Orientation of the level relative to our world frame is set by convention. Knowing which direction is "up" is important in a 3D game; in our case, we'll be using the z-axis, but the y-axis is also commonly used. Aligning the level to the other two axes (in our case, x and y) is arbitrary, but if our level is either gridlike or box-shaped, it is usually convenient to orient the grid lines or box sides to these remaining axes.

Positioning the level relative to the origin of the frame is also arbitrary but is usually set so that the origin lies in the center of a box defining our maximum play area. This helps avoid precision problems, since floating-point precision is centered around 0 (see Chapter 1). For example, we might have a 300-meter by 300-meter play area, so that in the xy directions the origin will lie directly in the center. While we can set things so that the origin is centered in z as well, we may want to adjust that depending on our application. If our game mainly takes place on a flat play area, such as in an arena fighting game, we might set the floor so that it lies at the origin; this will make it simple to place objects and characters exactly at floor level. In a submarine game, we might place sea level at the origin; negative z lies under the waterline and positive z above.

Placing Objects

If we were to use the objects' local coordinates directly in the world frame, they would end up interpenetrating and centered around the world origin. To avoid that situation, we apply affine transformations to each object to place them at their own specific position and orientation in the world. For each object, this is known as their particular *local-to-world transformation*. We often display the relative position and orientation of a particular object in the world by drawing its frame relative to the world frame (Figure 4.15). The local-to-world transformation, or world transformation for short, describes this relative relationship: The column vectors of the local-to-world matrix $\mathbf{A}$ describe where the local frame's basis vectors will lie relative to the world space basis, and the vector $\mathbf{y}$ describes where the local frame's origin lies relative to the world origin.

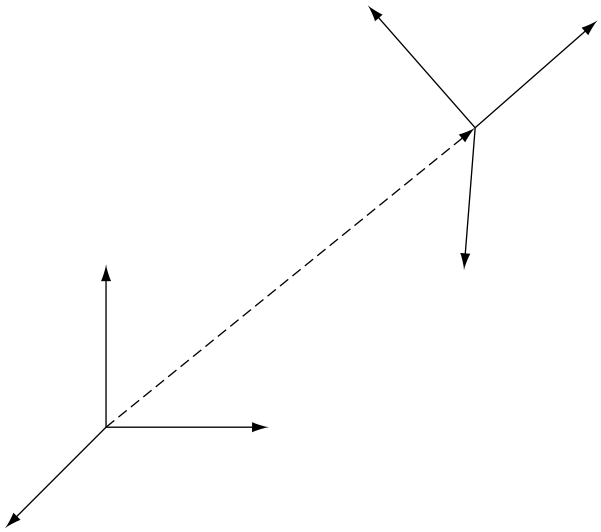


Figure 4.15 Local-to-world transformation.

The most commonly used affine transformations for object placement are translation, rotation, and scaling. Translation and rotation are convenient for two reasons. First, they correspond naturally to two of the characteristics we want to control in our objects: position and orientation. Second, they are rigid transformations, meaning they don't affect the size or shape of our object, which is generally the desired effect. Scaling is a deformation but is commonly useful to change the size of objects. For example, if two artists build two objects but fail to agree on a relative measure of size, you might end up with a table bigger than a room, if placed directly in the level. Rather than have the artist redo the model, we can use scaling to make it appear smaller. Scaling is also useful in fantastical games to either shrink a character to fit in a small space or grow a character to be more imposing. However, for most games you can actually get away without using scaling at all.

Source Code
Demo
Interaction

To create the final world transformation, we'll be concatenating a sequence of these translation, rotation, and scaling transformations together. However, remember that concatenation of transformations is not commutative. So, the order in which we apply our transformations affects the final result, sometimes in surprising ways. One basic example is transforming the point (0, 0, 0). A pure rotation around the origin has no effect on (0, 0, 0), so rotating by 90 degrees around z and then translating by (t_x, t_y, t_z) will just act as a translation, and we end up with (t_x, t_y, t_z). Translating the point first will transform it to (t_x, t_y, t_z), so in this case a subsequent rotation of 90 degrees around z will have an effect, with the final result

of $(-t_y, t_x, t_z)$. As another example, look at Figure 4.16(a), which shows a rotation and translation. Figure 4.16(b) shows the equivalent translation and rotation.

Scaling and rotation are also noncommutative. If we first scale (1, 0, 0) by (s_x, s_y, s_z), we get the point $(s_x, 0, 0)$. Rotating this by 90 degrees around z, we end up with $(0, s_x, 0)$. Reversing the transformation order, if we rotate (1, 0, 0) by 90 degrees around z, we get the point (0, 1, 0). Scaling this by (s_x, s_y, s_z), we get the point $(0, s_y, 0)$. Note that in the second case we rotated our object so that our original x-axis lies along the y-axis and then applied our scale, giving us the unexpected result. Figures 4.17(a) and 4.17(b) show another example of this applied to an object.

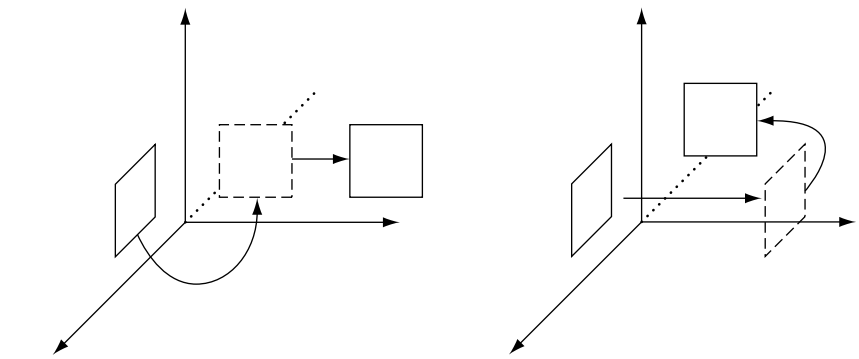


FIGURE 4.16 (a) Rotation, then translation and (b) translation, then rotation.

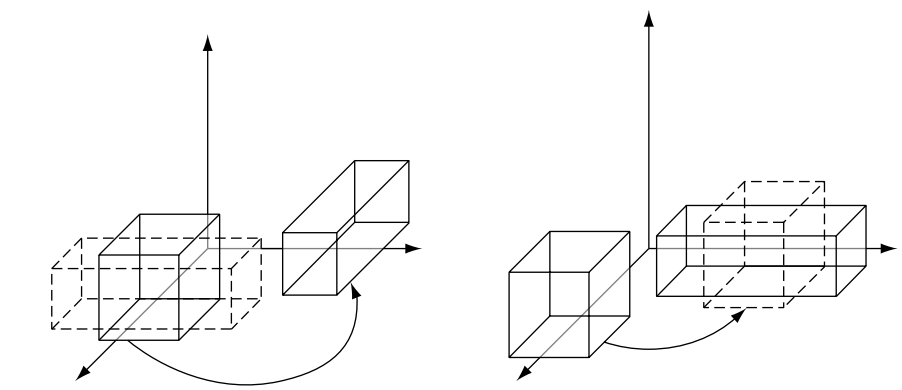


FIGURE 4.17 (a) Scale, then rotation and (b) rotation, then scale.

The final combination is scaling and translation. Again, this is not commutative. Remember that pure scaling is applied from the origin of the frame. If we translate an object from the origin and then scale, there will be additional scaling done to the translation of the object. So, for example, if we scale (1, 1, 1) by (s_x, s_y, s_z) and then translate by (t_x, t_y, t_z), we end up with $(t_x + s_x, t_y + s_y, t_z + s_z)$. If instead we translate first, we get $(t_x + 1, t_y + 1, t_z + 1)$, and then scaling gives us $(s_x t_x + s_x, s_y t_y + s_y, s_z t_z + s_z)$. Another example can be seen in Figures 4.18(a) and 4.18(b).

Generally, the desired order we wish to use for these transforms is to scale first, then rotate, then translate. Scaling first gives us the scaling along the axes we expect. We can then rotate around the origin of the frame, and then translate it into place. This gives us the following multiplication order:

$$\mathbf{M} = \mathbf{TRS}$$

4.4.2 Matrix Decomposition

It is sometimes useful to break an affine transformation matrix into its component basic affine transformations. This is called *matrix decomposition*. We performed one such decomposition when we pulled the translation information out of the matrix, effectively representing our transformation as the product of two matrices:

$$\begin{bmatrix} \mathbf{A} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{y} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Suppose we continue the process and break down **A** into the product of more basic affine transformations. For example, if we're using only scaling,

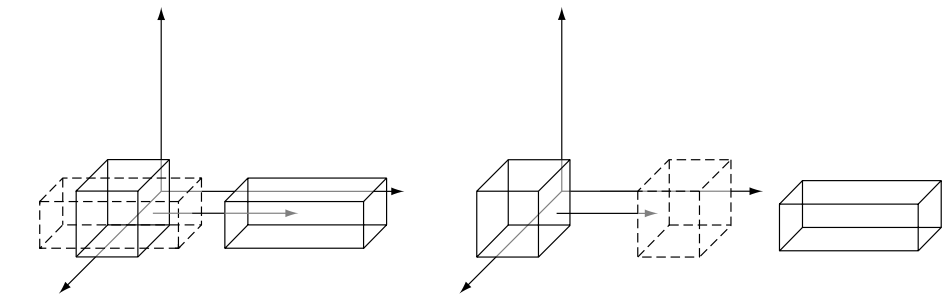


Figure 4.18 (a) Scale, then translation and (b) translation, then scale.

rotation, and translation, it would be ideal if we could break **A** into the product of a scaling and rotation matrix. If we know for a fact that **A** is the product of only a scaling and rotation matrix, in the order **RS**, we can multiply it out to get

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x r_{11} & s_y r_{12} & s_z r_{13} & 0 \\ s_x r_{21} & s_y r_{22} & s_z r_{23} & 0 \\ s_x r_{31} & s_y r_{32} & s_z r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this case, the lengths of the first three column vectors will give our three scale factors s_x, s_y, and s_z. To get the rotation matrix, all we need to do is normalize those three vectors.

Unfortunately, it isn't always that simple. As we'll see in Section 4.5, often we'll be concatenating a series of **TRS** transformations to get something like

$$\mathbf{M} = \mathbf{T}_n\mathbf{R}_n\mathbf{S}_n \cdots \mathbf{T}_1\mathbf{R}_1\mathbf{S}_1\mathbf{T}_0\mathbf{R}_0\mathbf{S}_0$$

In this case, even ignoring the translations, it is impossible to decompose **M** into the form **RS**. As a quick example, suppose that all these transformations with the exception of $\mathbf{S}_1$ and $\mathbf{R}_0$ are the identity transformation. This simplifies to

$$\mathbf{M} = \mathbf{S}_1\mathbf{R}_0$$

Now suppose $\mathbf{S}_1$ scales by 2 along y and by 1 along x and z, and $\mathbf{R}_0$ rotates by 60 degrees around z. Figure 4.19 shows how this affects a square on the xy plane. The sides of the transformed square are no longer perpendicular. Somehow, we have ended up applying a shear within our transformation, and clearly we cannot represent this by a simple concatenation **RS**.

One solution is to decompose the matrix using a technique known as *singular value decomposition*, or simply SVD. Assuming no translation, the matrix **M** can be represented by three matrices **L**, **D**, and **R**, where **L** and **R** are orthogonal matrices, **D** is a diagonal matrix with nonnegative entries, and

$$\mathbf{M} = \mathbf{LDR}$$

An alternative formulation to this is *polar decomposition*, which breaks the nontranslational part of the matrix into two pieces: an orthogonal matrix **Q** and a stretch matrix **S**, where

$$\mathbf{S} = \mathbf{U}^T\mathbf{KU}$$

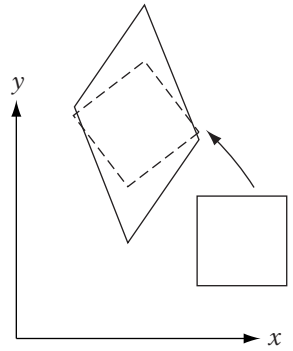


FIGURE 4.19 Effect of rotation, then scale.

Matrix **U** in this case is another orthogonal matrix, and **K** is a diagonal matrix. The stretch matrix combines the scale-plus-shear effect we saw in our example: It rotates the frame to an orientation, scales along the axes, and then rotates back. Using this, a general affine matrix can be broken into four transformations:

$$\mathbf{M} = \mathbf{TRNS}$$

where **T** is a translation matrix, **Q** has been separated into a rotation matrix **R** and a reflection matrix $\mathbf{N} = \pm\mathbf{I}$, and **S** is the preceding stretch matrix.

Performing either SVD or polar decomposition is out of the purview of this text. As we'll see, there are ways to avoid matrix decomposition at the cost of some conversion before we send our models down the graphics pipeline. However, at times we may get a matrix of unknown structure from a library module that we don't control. For example, we could be using a commercial physics engine or writing a plug-in for a 3D modeling package such as Max or Maya. Most of the time a function is provided that will decompose such matrices for us, but this isn't always the case. For those times and for those who are interested in pursuing this topic, more information on decompositions can be found in Goldman [42], Golub and Van Loan [44], and Shoemake and Duff [105].

4.4.3 Avoiding Matrix Decomposition

In the preceding section we made no assumptions about the values for our scaling factors. Now let's assume that they are equal; that is, each scaling matrix performs a uniform scale. Looking at just the rotation and scaling transformations, we have

$$\mathbf{M} = \mathbf{R}_n\mathbf{S}_n \cdots \mathbf{R}_1\mathbf{S}_1\mathbf{R}_0\mathbf{S}_0$$

Since each scaling transformation is uniformly scaling, we can simplify this to

$$\mathbf{M} = \mathbf{R}_n\sigma_n \cdots \mathbf{R}_1\sigma_1\mathbf{R}_0\sigma_0$$

Using matrix algebra, we can shuffle terms to get

$$\begin{aligned}\mathbf{M} &= \mathbf{R}_n \cdots \mathbf{R}_1\mathbf{R}_0\sigma_n \cdots \sigma_1\sigma_0 \\ &= \mathbf{R}\sigma \\ &= \mathbf{RS}\end{aligned}$$

where $\mathbf{R}$ is a rotation matrix and $\mathbf{S}$ is a uniform scaling matrix. So, if we use uniform scaling, we can in fact decompose our matrix into a rotation and scaling matrix, as we just did.

However, even in this case the decomposition takes three square roots and nine scaling operations to perform. This leads to an alternate approach to handling transformations. Instead of storing transformations for our objects as a single 4×4 or even 3×4 matrix, we will break out the individual parts: a scale factor s, a 3×3 rotation matrix $\mathbf{R}$, and a translation vector $\mathbf{t}$. To apply this transformation to a point P, we use

$$\mathcal{T}(P) = \begin{bmatrix} s\mathbf{R}\mathbf{x} + \mathbf{t} \\ 1 \end{bmatrix}$$

Note the similarity to equation 4.1. We've replaced $\mathbf{A}$ with $s\mathbf{R}$ and $\mathbf{y}$ with $\mathbf{t}$. In practice we ignore the trailing 1.

Concatenating transformations in matrix format is as simple as performing a multiplication. Concatenating in our alternate format is a little less straightforward but is not difficult and actually takes fewer operations on a standard floating-point processor:

$$\begin{aligned} s' &= s_1 s_0 \\ \mathbf{R}' &= \mathbf{R}_1\mathbf{R}_0 \\ \mathbf{t}' &= \mathbf{t}_1 + s_1\mathbf{R}_1\mathbf{t}_0 \end{aligned} \tag{4.16}$$

Computing the new scale and rotation makes a certain amount of sense, but it may not be clear why we don't add the two translations together to get the new translation. If we multiply the two transforms in matrix format, we have the following order:

$$\mathbf{M} = \mathbf{T}_1\mathbf{R}_1\mathbf{S}_1\mathbf{T}_0\mathbf{R}_0\mathbf{S}_0$$

But since $\mathbf{T}_0$ is applied after $\mathbf{R}_0$ and $\mathbf{S}_0$, they have no effect on it. So, if we want to find how the translation changes, we drop them:

$$\mathbf{M}' = \mathbf{T}_1 \mathbf{R}_1 \mathbf{S}_1 \mathbf{T}_0$$

Multiplying this out in block format gives us

$$\begin{aligned}
\mathbf{M}' &= \begin{bmatrix} \mathbf{I} & \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} s_1\mathbf{I} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{t}_0 \\ \mathbf{0}^T & 1 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} s_1\mathbf{I} & s_1\mathbf{t}_0 \\ \mathbf{0}^T & 1 \end{bmatrix} \\
&= \begin{bmatrix} s_1\mathbf{R}_1 & s_1\mathbf{R}_1\mathbf{t}_0 + \mathbf{t}_1 \\ \mathbf{0}^T & 1 \end{bmatrix}
\end{aligned}$$

We can see that the right-hand column vector $\mathbf{y}$ is equal to equation 4.16. To get the final translation we need to apply the second scale and rotation before adding the second translation. Another way of thinking of this is that we need to scale and rotate the first translation vector into the frame of the second translation vector before they can be combined together.

There are a few advantages to this alternate format. First of all, it's clear what each part does — the scale and rotation aren't combined into a single 3×3 matrix. Because of this, it's also easier to change individual elements. We can update rotation, scale through a simple multiplication, or even just set them directly. Surprisingly, on a serial processor concatenation is also cheaper. It takes 48 multiplications and 32 adds to do a traditional matrix multiplication, but only 40 multiplications and 27 adds to perform our alternate concatenation. This advantage disappears when using vector processor operations, however. In that case, it's much easier to parallelize the matrix multiplication (16 operations on some systems), and the cost of scaling and rotating the translation vector becomes more of an issue.

Even with serial processors our alternate format does have one main disadvantage, which is that we need to create a 4×4 matrix to be sent to the graphics application programming interface (API). Based on our previous explorations of the transformation matrix, we can create a matrix from our alternate format quite quickly; scale the three columns of the rotation matrix; and then copy it and the translation vector into our 4×4:

$$\begin{bmatrix} sr_{0,0} & sr_{0,1} & sr_{0,2} & t_x \\ sr_{1,0} & sr_{1,1} & sr_{1,2} & t_y \\ sr_{2,0} & sr_{2,1} & sr_{2,2} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which representation is better? It depends on your application. If all you wish to do is an initial scale and then apply sequences of rotations and translations, the 4×4 matrix format works fine and will be faster on a vector processor. If, on the other hand, you wish to make changes to scale as well, using the alternate format should at least be considered. And, as we'll see, if we wish to use a rotation representation other than a matrix, the alternate formation is almost certainly the way to go.

4.5 Object Hierarchies

In describing object transformations, we have considered them as transforming from the object's local frame (or local space) to a world frame (or world space). However, it is possible to define an object's transformation as being relative to another object's space instead. We could carry this out for a number of steps, thereby creating a hierarchy of objects, with world space as the root and each object's local space as a node in a tree (Figure 4.20).

For example, suppose we wish to attach an arm to a body. The body is built with its origin relative to its center. The arm has its origin at the shoulder joint location because that will be our center of rotation. If we were to place them in the world using the same transformation, the arm would end up inside the body instead of at the shoulder. We want to find the transformation that modifies the arm's world transformation so that it matches the movement of the body and still remains at the shoulder. The way to do this is to define a transformation for the arm relative to the body's local space. If we combine this with the transformation for the body, this should place the arm in the correct place in world space relative to the body, no matter its position and orientation.

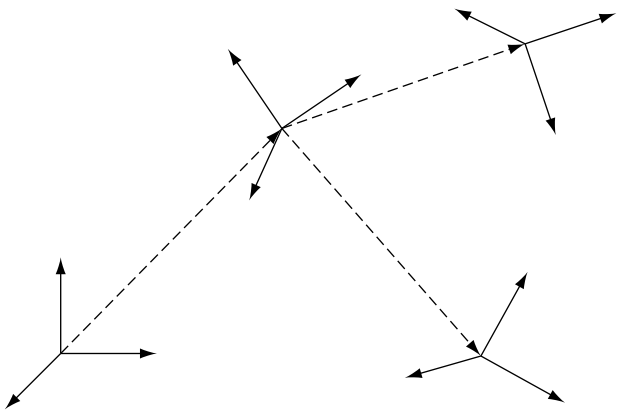
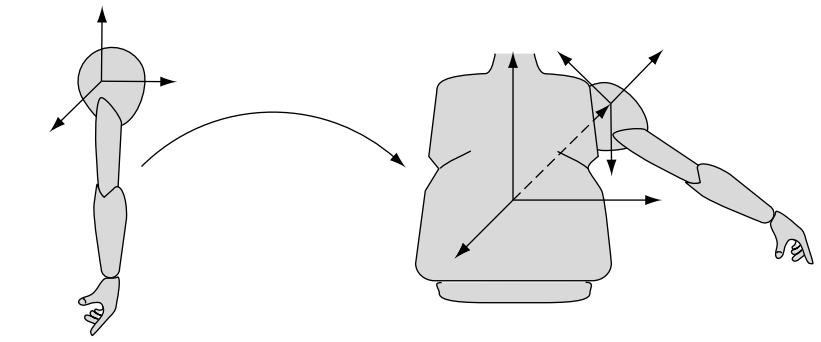


Figure 4.20 Hierarchy of frames.

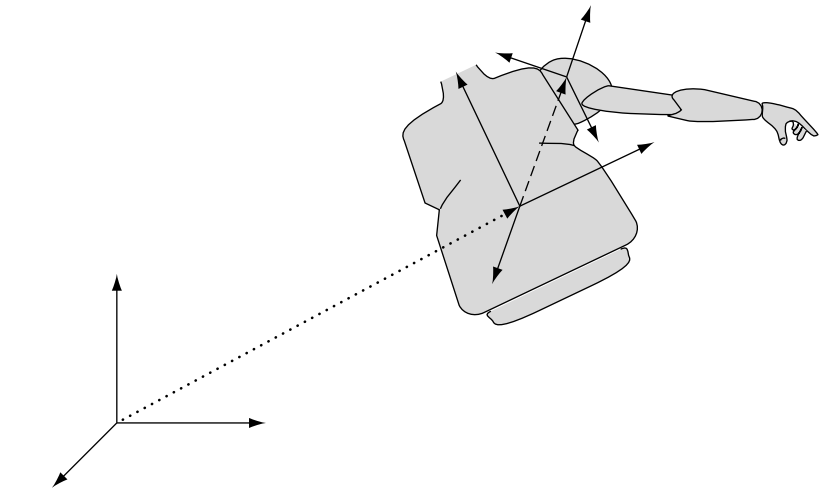
The idea is to transform the arm to body space (Figure 4.21(a)) and then continue the transform into world space (Figure 4.21(b)). In this case, for each stage of transformation we perform the order as scale, rotate, and then translate. In matrix format the world transformation for the arm would be

$$\mathbf{W} = \mathbf{T}_{body}\mathbf{R}_{body}\mathbf{S}_{body}\mathbf{T}_{arm}\mathbf{R}_{arm}\mathbf{S}_{arm}$$

As we've indicated, the body and arm are treated as two separate objects, each with its own transformations, placed in a hierarchy. The body transformation is relative to world space, and the arm transformation is relative to the



(a)



(b)

FIGURE 4.21 (a) Mapping arm to body's local space and (b) mapping body and arm to world space.

body's space. When rendering, for example, we begin by drawing the body with its world transformation and then drawing the arm with the concatenation of the body's transformation and the arm's transformation. By doing this, we can change them independently — rotating the arm around the shoulder, for example, without affecting the body at all. Similar techniques can be used to create deeper hierarchies, for example, a turret that rotates on top of a tank chassis, with a gun barrel that elevates up and down relative to the turret.

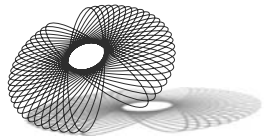
One way of coding this is to create separate objects, each of which handles all the work of grabbing the transformation from the parent objects and combining to get the final display transform. The problem with this approach is that it generates a lot of duplicated code. Using the tank example, the code necessary for handling the hierarchy for the turret is going to be almost identical to that for the barrel. What is usually done is to design a data structure that handles the generalized case of a hierarchy of frames and use that to manage our hierarchical objects. We've implemented an example using one such data structure called a *scene graph*. More detail about this example and scene graphs in general can be found on the CD-ROM.

4.6 Chapter Summary

In this chapter we've discussed the general properties of affine transformations, how they map between affine spaces, and how they can be represented and performed by matrices at one dimension higher than the affine spaces involved. We've covered the basic affine transformations as used in interactive applications and how to combine three of them — scaling, rotation, and translation — to manipulate our objects within our world. While it may be desirable to separate a given affine transformation back into scaling, rotation, and translation components, we have seen that it is not always possible when using nonuniform scaling. Separating components in this manner may not be efficient, so we have presented an alternative affine transformation representation with the three components separated. Finally, we have discussed how to construct transformations relative to other objects, which allows us to create jointed, hierarchical structures.

For those interested in reading further, information on affine algebra can be found in Schneider and Eberly [100], as well as in deRose [23]. The standard affine transformations are described in most graphics textbooks, such as Möller and Haines [82] and Foley et al. [38]. Further details on hierarchical transformation management and scene graph construction and usage can be found in Eberly [25].

This page intentionally left blank

Chapter 5

Orientation Representation

5.1 Introduction

In the previous chapter we discussed various types of affine transformations and how they can be represented by a matrix. In this chapter we will focus specifically on orientation and the rotation transformation. We'll look at four different orientation formats and compare them on the basis of the following criteria:

- Represents orientation/rotation with a small number of values.
- Can be concatenated efficiently to form new orientations/rotations.
- Rotates points and vectors efficiently.

The first item is important if memory usage is an issue, either because we are working with a memory-limited machine such as a console, or because we want to store a large number of transformations, such as in animation data. In either case, any reduction in representation size means that we have freed-up memory that can be used for more animations, for more animation frames (leading to a smoother result), or for some other aspect of the game. Rotating points and vectors efficiently may seem like an obvious requirement, but one that merits mentioning; not all representations are good at this. Similarly, for some representations concatenation is not possible.

There are two other criteria we might consider for an orientation format that we will not discuss here: how well the representation can be interpolated

and how suitable it is for numeric integration in physics. Both of these topics will be discussed in Chapters 10 and 13, respectively.

As we'll see, there is no one choice that meets all of our requirements; each has its strengths and weaknesses in each area, depending on our implementation needs.

5.2 Rotation Matrices

Since we have been using matrices as our primary orientation/rotation representation, it is natural to begin our discussion with them.

For our first desired property, memory usage, matrices do not fare well. Euler's rotation theorem states that the minimum number of values needed to represent a rotation in three dimensions is three. The smallest possible rotation matrix requires nine values, or three orthonormal basis vectors. It is possible to compress a rotation matrix, but in most cases this is not done unless we're sending data across a network. Even then it is better to convert to one of the more compact representations that we will present in the following sections, rather than compress the matrix.

However, for the second two properties, matrices do quite well. Concatenation is done through a matrix-matrix multiplication, and rotating a vector is done through a matrix-vector multiplication. Both of these are reasonably efficient on a standard floating-point processor. But on a processor that supports SSE or Altivec instructions, which can perform matrix and vector operations in parallel, both of these operations can be performed even faster. Most graphics hardware has built-in circuitry that performs similarly. And as we've seen, 4×4 matrices can be useful for more than just rotation. Because of all these reasons, matrices continue to be useful despite their memory footprint.

5.3 Fixed and Euler Angles

5.3.1 Definition

We've just stated that the minimum number of values needed to represent a rotation in three-dimensional (3D) space is three. As it happens, these three values can be the angles of three sequential rotations around a set of orthogonal axes. In Chapter 4, we used this as one means of building a generalized rotation matrix. Our chosen sequence of axes in this case was z-y-x, so the values $(0, \pi/4, \pi/2)$ represent a rotation of 0 radians around the z-axis, followed by a rotation of $\pi/4$ radians (or 45 degrees) around the y-axis, and concluding with a rotation of $\pi/2$ radians (90 degrees) around the x-axis. Angles can be less than 0 or greater than 2π, to represent reversed rotations

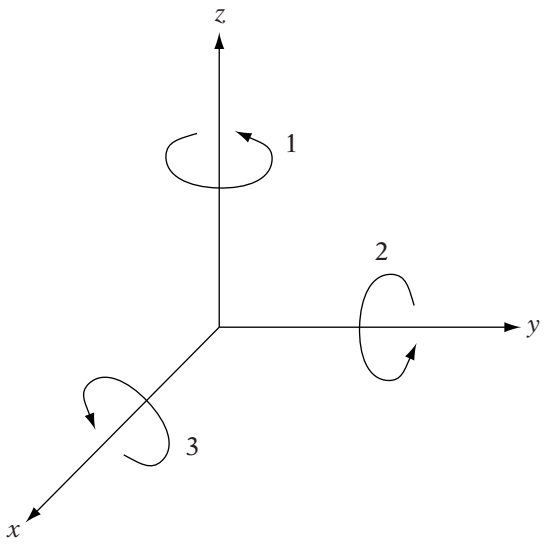


FIGURE 5.1 Order and direction of rotation for *z*-*y*-*x* fixed angles.

and multiple rotations around a given axis. Note that we are using radians rather than degrees to represent our angles; either convention is acceptable, but the trigonometric functions used in C or C++ expect radians.

The order we've given is somewhat arbitrary, as there is no standard order that is used for the three axes. We could have used the sequence *x*-*y*-*z* or *z*-*x*-*y* just as well. We can even duplicate one axis, so long as it is not the same axis in a row, so *y*-*z*-*y* is a valid sequence, while an axis rotation sequence such as *z*-*y*-*y* is not permitted. This is because duplicating an axis is redundant and doesn't add an additional degree of freedom.

These rotations are performed around either the world axes or the object's model axes. When the angles represent world axis rotations, they are usually called *fixed angles* (Figure 5.1). The most convenient way to use fixed angles is to create an *x*-, *y*-, or *z*-rotation matrix for each angle and apply it in turn to our set of vertices. So an *x*-*y*-*x* fixed-angle representation can be concatenated into a single matrix $\mathbf{R} = \mathbf{R}_x \mathbf{R}_y \mathbf{R}_x$ in matrix form.

A sequence of model axis rotations, in turn, is said to consist of *Euler angles*.[1] The three Euler angles are commonly known as *roll*, *pitch*, and *heading*, after the three axes in a ship or an airplane. Heading is also sometimes referred to as *yaw*. Roll represents rotation around the forward axis, pitch rotation around a side axis, and heading rotation around the up axis

1. Just to be confusing, sometimes (a sequence of) rotations around world space axes are also referred to as Euler angles. Hopefully context will tell you which one the author means.

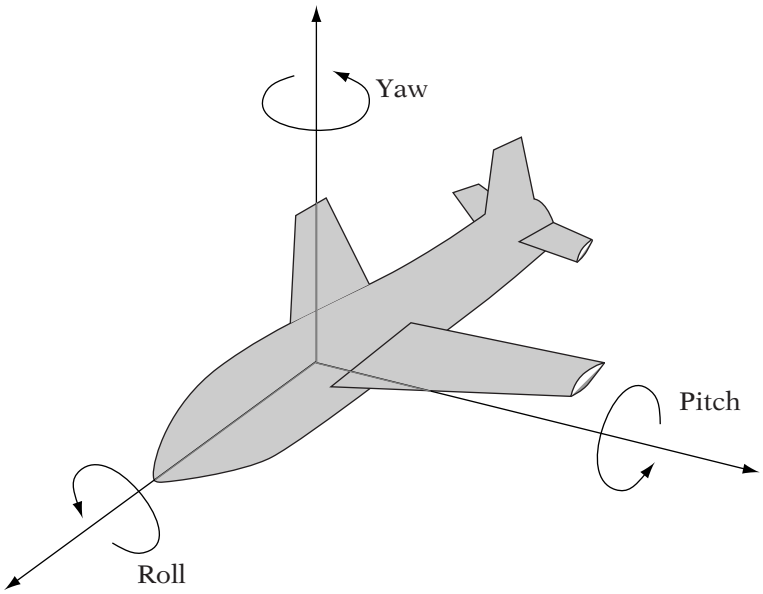


FIGURE 5.2 Roll, pitch, and rotations relative to model coordinate axes.

(Figure 5.2). Whether a given roll, pitch, or heading rotation is around x, y, or z depends on how we've defined our coordinate frame. Suppose we are using a coordinate system where the z-axis represents up, the x-axis represents forward, and the y-axis represents left. Then heading is rotation around the z-axis, pitch is rotation around the y-axis, and roll is rotation around the x-axis. They are commonly applied in the order roll-pitch-heading, so the corresponding Euler angles for our case are x-y-z.

To create a rotation matrix that applies Euler angles, we concatenate in the reverse order of fixed angles. To see why, let's take our set of x-y-z Euler angles. We begin by applying the $\mathbf{R}_x$ matrix, to give us a rotation around x. We then want to apply a rotation around the object's initial model y-axis. However, because of the x rotation, the y-axis has been transformed to a new orientation. So, if we concatenate as we normally would, our rotation will be about the transformed y-axis, which is not what we want. To avoid this, we transform by $\mathbf{R}_y$ first, then by $\mathbf{R}_x$, giving $\mathbf{R}_x\mathbf{R}_y$. The same is true for the z rotation: We need to rotate around z first to ensure we rotate around the original model z-axis, not the transformed one. The resulting matrix is

$$\mathbf{R}_{Euler} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z$$

So x-y-z Euler angles are the same as z-y-x fixed angles.

5.3.2 Format Conversion

By concatenating three general axis rotation matrices and expanding out the terms, we can create a generalized rotation matrix. The particular matrix will depend on which axis rotations we're using and whether they are fixed or Euler angles. For *z*-*y*-*x* fixed angles or *x*-*y*-*z* Euler angles, the matrix looks like

$$\mathbf{R} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z = \begin{bmatrix} CyCz & -CySz & Sy \\ SxSyCz + CxSz & -SxSySz + CxCz & -SxCy \\ -CxSyCz + SxSz & CxSySz + SxCz & CxCy \end{bmatrix}$$

where

$$\begin{aligned} Cx &= \cos\theta_x & Sx &= \sin\theta_x \\ Cy &= \cos\theta_y & Sy &= \sin\theta_y \\ Cz &= \cos\theta_z & Sz &= \sin\theta_z \end{aligned}$$

This should look familiar from Chapter 4.

When possible, we can save some instructions by computing each sine and cosine using a single `sincos()` call. This function is not supported on all processors, or even in all math libraries, so we have provided a wrapper function `IvSinCosf()` (accessible by including `IvMath.h`) that will calculate it depending on the platform.

We can convert from a matrix back to a possible set of fixed angles by inverting this process. Note that since we'll be using inverse trigonometric functions there are multiple resulting angles. We'll also be taking a square root, the result of which could be positive or negative. Hence, there are multiple possibilities of Euler or fixed angles for a given matrix—the best we can do is find one. Assuming we're using *z*-*y*-*x* fixed angles, we can see that $\sin\theta_y$ is equal to $\mathbf{R}_{02}$. Finding $\cos\theta_y$ can be done by using the identity $\cos\theta_y = \sqrt{1 - \sin^2\theta_y}$. The rest falls out from dividing quantities out of the first row and last column of the matrix, so

$$\begin{aligned} \sin\theta_y &= \mathbf{R}_{02} \\ \cos\theta_y &= \sqrt{1 - \sin^2\theta_y} \\ \sin\theta_x &= -\mathbf{R}_{12}/\cos\theta_y \\ \cos\theta_x &= \mathbf{R}_{22}/\cos\theta_y \\ \sin\theta_z &= -\mathbf{R}_{01}/\cos\theta_y \\ \cos\theta_z &= \mathbf{R}_{00}/\cos\theta_y \end{aligned}$$

Note that we have no idea whether $\cos\theta_y$ should be positive or negative, so we assume that it's positive. Also, if $\cos\theta_y = 0$, then the x and z axes have become aligned (see Section 5.3.5) and we can't distinguish between rotations around x and rotations around z. One possibility is to assume that rotation around z is 0, so

$$\sin\theta_z = 0$$
$$\cos\theta_z = 1$$
$$\sin\theta_x = \mathbf{R}_{21}$$
$$\cos\theta_x = \mathbf{R}_{11}$$

Calling `arctan2()` for each sin/cos pair will return a possible angle in radians, generally in the range $[-\pi, \pi]$. Note that we have lost one of the few benefits of fixed and Euler angles, which is that they can represent multiple rotations around an axis by using angles greater than 2π radians, or 360 degrees. We have also lost any notion of "negative" rotation.

5.3.3 Concatenation

Clearly, fixed and Euler angles meet our first criteria for a good orientation representation: They use the minimum number of values. However, they don't really meet the remainder of our requirements. First of all, they don't concatenate well. Adding angles doesn't work: Applying $(\pi/2, \pi/2, \pi/2)$ twice doesn't end up at the same orientation as (π, π, π). The most straightforward method for concatenating two Euler or fixed-angle triples is to convert each sequence of angles to a matrix, concatenate the matrix, and then convert the matrix back to Euler or fixed angles. This will take a large number of operations, and will only give an approximate result, due to the ill-formed nature of the matrix to fixed and Euler conversion.

5.3.4 Vector Rotation

Euler and fixed angles also aren't the most efficient method for rotating vectors. Recall that to rotate a vector around z uses the formula

$$R_z(x, y, \theta) = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$$

Using the angles directly means that for each axis, we compute a sine and cosine and then apply the preceding formula. Even if we cache the sine and cosine values for a set of vectors, this ends up being more expensive than the cost of a matrix multiplication. Therefore, when rotating multiple vectors

(in general the break-even point is five vectors), it's more efficient to convert to matrix format.

5.3.5 Other Issues

As if all of these disadvantages are not enough, the fatal blow is that in certain cases fixed or Euler angles can lose one degree of freedom. We can think of this as a mathematical form of *gimbal lock*. In aeronautic navigational systems, there is often a set of gyroscopes, or gimbals, that control the orientation of an airplane or rocket. Gimbal lock is a mechanical failure where one gimbal is rotated to the end of its physical range and it can't be rotated any further, thereby losing one degree of freedom. While in the virtual world, we don't have mechanical gyroscopes to worry about, a similar situation can arise.

Suppose we are using x-y-z fixed angles and we consider the case where, no matter what we use for the x and z angles, we will always rotate around the y-axis by 90 degrees. This rotates the original world x-axis — the axis we first rotate around — to be aligned with the world negative z-axis (Figure 5.3). Now any rotation we do with θ_z will subtract from any rotation to which we have applied θ_x. The combination of x and z rotations can be represented by one value $\theta_x - \theta_z$, applied as the initial x-axis rotation. For example, in Figure 5.4, applying the fixed angles $(\pi/2, \pi/2, \pi/2)$ gets us back to our original $(0, \pi/2, 0)$. Instead of using $(\theta_x, \pi/2, \theta_z)$, we could just as well use $(\theta_x - \theta_z, \pi/2, 0)$

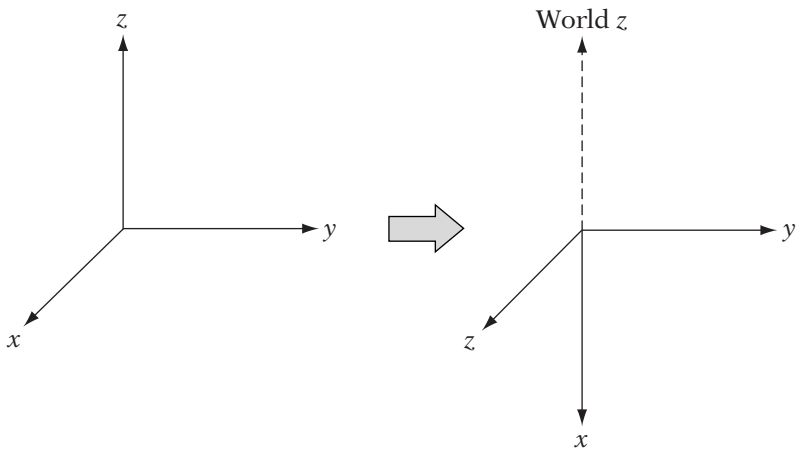


Figure 5.3 Demonstration of mathematical gimbal lock. A rotation of 90 degrees around y will lead to the local x-axis aligning with the $-z$ world axis, and a loss of a degree of freedom.

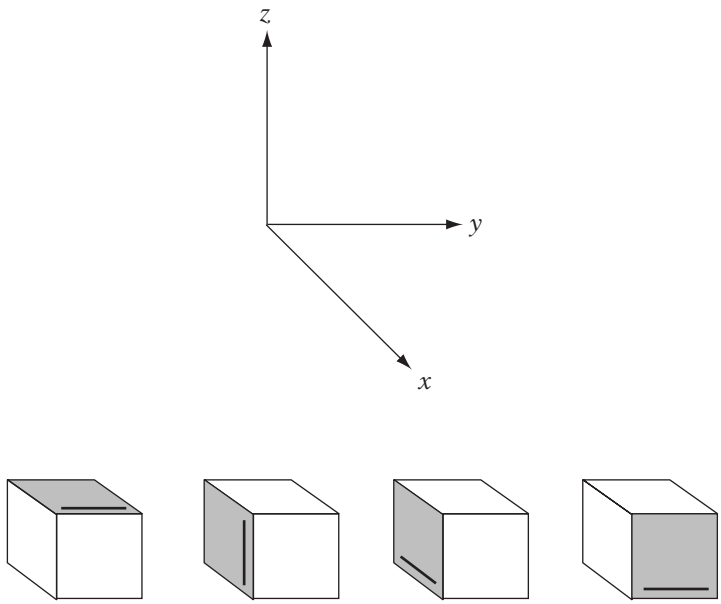


FIGURE 5.4 Effect of gimbal lock. Rotating the box around the world x-axis, then the world y-axis, then the world z-axis ends up having the same effect as rotating the box around just the y-axis.

or $(0, \pi/2, \theta_z - \theta_x)$. Another way to think of this is: Were this in matrix form we would not be able to extract unique values for θ_x and θ_z. We have effectively lost one degree of freedom.

To try this for yourself, take an object whose orientation can be clearly distinguished, like a book or CD case. From your point of view, rotate the object clockwise 90 degrees around an axis pointing forward (roll). Now rotate the new top of the object away from you by 90 degrees (pitch). Now rotate the object counterclockwise 90 degrees around an axis pointing up (heading). The result is the same as pitching the object downward 90 degrees (see Figure 5.4).

Still, in some cases fixed or Euler angles do provide an intuitive representation for orientation. For example, in a hierarchical system it is very intuitive to define rotations at each joint as a set of Euler angles and to constrain certain axes to remain fixed. An elbow or knee joint, for instance, could be considered a set of Euler angles with two constraints and only one axis available for applying rotation. It's also easy to set a range of angles so that the joint doesn't bend too far one way or the other. However, these limited advantages are not enough to outweigh the problems with fixed and Euler angles. So in most cases, fixed and Euler angles are used as a means to semi-intuitively set

other representations (being aware of the dangers of gimbal lock, of course), and our library will be no exception.

5.4 Axis–Angle Representation

5.4.1 Definition

Recall from Chapter 4 that we can represent a general rotation in $\mathbb{R}^3$ by an axis of rotation, and the amount we rotate around this axis by an angle of rotation. Therefore, we can represent rotations in two parts: a 3-vector $\mathbf{r}$ that lies along the axis of rotation, and a scalar θ that corresponds to a counterclockwise rotation around the axis, if the axis is pointing toward us. Usually, a normalized vector $\hat{\mathbf{r}}$ is used instead, which constrains the four values to three degrees of freedom, corresponding to the three degrees of freedom necessary for 3D rotations.

Generating the axis–angle rotation that takes us from one normalized vector $\hat{\mathbf{v}}$ to another vector $\hat{\mathbf{w}}$ is straightforward (Figure 5.5). The angle of rotation is the angle between the two vectors:

$$\theta = \arccos(\hat{\mathbf{v}} \cdot \hat{\mathbf{w}}) \tag{5.1}$$

The two vectors lie in the plane of rotation, and so the axis of rotation is perpendicular to both of them:

$$\mathbf{r} = \hat{\mathbf{v}} \times \hat{\mathbf{w}} \tag{5.2}$$

Normalizing $\mathbf{r}$ gives us $\hat{\mathbf{r}}$. Near-parallel vectors may cause us some problems either because the dot product is near 0, or normalizing the cross product

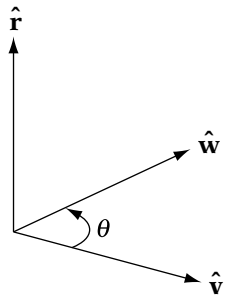


Figure 5.5 Axis–angle representation. Rotation by $\mathbf{r}$ by angle θ rotates $\mathbf{v}$ into $\mathbf{w}$.

ends up dividing by a near-zero value. In those cases, we set θ to 0 and $\hat{\mathbf{r}}$ to any arbitrary, normalized vector.

5.4.2 Format Conversion

To convert an axis–angle represention to a matrix, we can use the derivation from Chapter 4:

$$\mathbf{R}_{\hat{\mathbf{r}}\theta} = \begin{bmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + c \end{bmatrix} \tag{5.3}$$

where

$$\begin{aligned} \hat{\mathbf{r}} &= (x, y, z) \\ c &= \cos\theta \\ s &= \sin\theta \\ t &= 1 - \cos\theta \end{aligned}$$

Converting from a matrix to the axis–angle format has similar issues as the fixed-angle format, since opposing vectors $\hat{\mathbf{r}}$ and $-\hat{\mathbf{r}}$ can be used to generate the same rotation by rotating in opposite directions, and multiple angles (0 and 2π, for example) applied to the same axis can rotate to the same orientation. The following method is from Eberly [26].

We begin by computing the angle. The sum of the diagonal elements, or *trace* of a rotation matrix $\mathbf{R}$, is equal to $2\cos\theta + 1$, where θ is our angle of rotation. This gives us an easy method for computing θ:

$$\theta = \arccos\left(\frac{1}{2}(\text{trace}(\mathbf{R}) - 1)\right)$$

There are three possibilities for θ. If θ is 0, then we can use any arbitrary unit vector as our axis. If θ lies in the range $(0, \pi)$, then we can compute the axis by using the formula

$$\mathbf{R} - \mathbf{R}^T = 2\sin\theta\mathbf{S} \tag{5.4}$$

where $\mathbf{S}$ is a skew symmetric matrix of the form

$$\mathbf{S} = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix}$$

The values x, y, and z in this case are the components of our axis vector $\hat{\mathbf{r}}$. We can compute $\mathbf{r}$ as $(R_{21} - R_{12}, R_{02} - R_{20}, R_{10} - R_{01})$, and normalize to get $\hat{\mathbf{r}}$.

If θ equals π, then $\mathbf{R} - \mathbf{R}^T = \mathbf{0}$, which doesn't help us at all. In this case, we can use another formulation for the rotation matrix, which only holds if $\theta = \pi$:

$$\mathbf{R} = \mathbf{I} + 2\mathbf{S}^2 = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy & 2xz \\ 2xy & 1 - 2x^2 - 2z^2 & 2yz \\ 2xz & 2yz & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

The idea is that we can use the diagonal elements to compute the three axis values. By subtracting appropriately, we can solve for one term, and then use that value to solve for the other two. For example, $R_{00} - R_{11} - R_{22} + 1$ expands to

$$\begin{aligned} R_{00} - R_{11} - R_{22} + 1 &= 1 - 2y^2 - 2z^2 - 1 + 2x^2 + 2z^2 - 1 + 2x^2 + 2y^2 + 1 \\ &= 4x^2 \end{aligned}$$

So,

$$x = \frac{1}{2}\sqrt{R_{00} - R_{11} - R_{22} + 1} \tag{5.5}$$

and consequently,

$$\begin{aligned} y &= \frac{R_{01}}{2x} \\ z &= \frac{R_{02}}{2x} \end{aligned}$$

To avoid problems with numeric precision and square roots of negative numbers, we'll choose the largest diagonal element as the term that we'll solve for. So, if R_{00} is the largest diagonal element, we'll use the preceding equations. If R_{11} is the largest, then

$$\begin{aligned} y &= \frac{1}{2}\sqrt{R_{11} - R_{00} - R_{22} + 1} \\ x &= \frac{R_{01}}{2y} \\ z &= \frac{R_{12}}{2y} \end{aligned}$$

Finally, if R_{22} is the largest element we use

$$z = \frac{1}{2}\sqrt{R_{22} - R_{00} - R_{11} + 1}$$

$$x = \frac{R_{02}}{2z}$$

$$y = \frac{R_{12}}{2z}$$

5.4.3 Concatenation

Concatenating two axis–angle representations is not straightforward. One method is to convert them to two matrices or two quaternions (see below), multiply, and then convert back to the axis–angle format. As one can easily see, this is more expensive than just concatenating two matrices. Because of this, one doesn't often perform this operation on axis–angle representations.

5.4.4 Vector Rotation

For the rotation of a vector $\mathbf{v}$ by the axis–angle representation $(\hat{\mathbf{r}}, \theta)$, we can use the Rodrigues formula that we derived in Chapter 4:

$$R\mathbf{v} = \cos\theta\,\mathbf{v} + [1 - \cos\theta](\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}} \times \mathbf{v})$$

If we precompute $\cos\theta$ and $\sin\theta$ and reuse intermediary values, we can compute this relatively efficiently. We can improve this slightly by using the identity

$$\begin{aligned}\hat{\mathbf{r}} \times (\hat{\mathbf{r}} \times \mathbf{v}) &= (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} - (\hat{\mathbf{r}} \cdot \hat{\mathbf{r}})\mathbf{v} \\ &= (\mathbf{v} \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} - \mathbf{v}\end{aligned}$$

and substituting to get an alternate Rodrigues formula:

$$R\mathbf{v} = \mathbf{v} + (1 - \cos\theta)[\hat{\mathbf{r}} \times (\hat{\mathbf{r}} \times \mathbf{v})] + \sin\theta(\hat{\mathbf{r}} \times \mathbf{v})$$

In both these cases, the trade-off is whether to store the results of the transcendental functions and thereby use more memory, or compute them every time and lose speed. The answer will depend on the needs of the implementation.

When rotating two or more vectors, it is more efficient to convert the axis–angle format to a matrix and then multiply. The break-even point is two vectors, so if you're only transforming one vector, don't bother converting; otherwise, use a matrix.

5.4.5 Axis–Angle Summary

While being a useful way of thinking about rotation, the axis–angle format still has some problems. Concatenating two axis–angle representations is extremely expensive. And unless we store two additional values, rotating vectors requires computing transcendental functions, which is not very efficient either. Our next representation encapsulates some of the useful properties of the axis–angle format, while providing a more efficient method for concatenation. It precomputes the transcendental functions and uses them to rotate vectors in nearly equivalent time to the axis–angle method. Because of this, we have not explicitly provided an implementation in our library for the axis–angle format.

5.5 Quaternions

5.5.1 Definition

Source Code
Library
IvMath
Filename
IvQuat

The final orientation representation we'll consider could be considered a variant of the axis–angle representation, and in fact when using it for rotation it's often simplest to think of it that way. It is called the *quaternion* and was created by the Irish mathematician Sir William Hamilton [52] in the nineteenth century and introduced to computer graphics by Ken Shoemake [103] in the 1980s. Quaternions require only four values, they don't have problems of gimbal lock, the mathematics for concatenation are relatively simple, and if properly constructed they can be used to rotate vectors in a reasonably efficient manner.

Hamilton's general formula for a quaternion $\mathbf{q}$ is as follows:

$$\mathbf{q} = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

The quantities $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ can be thought of as the standard basis for all quaternions, so it is common to write a quaternion as just

$$\mathbf{q} = (w, x, y, z)$$

The $x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$ part of the quaternion is akin to a vector in $\mathbb{R}^3$, so a quaternion also can be written as

$$\mathbf{q} = (w, \mathbf{v})$$

where w is called the scalar part and $\mathbf{v}$ is called the vector part.

Frequently, we'll want to use vectors in combination with quaternions. To do so, we'll zero out the scalar part and set the vector part equal to our original vector. So, the quaternion corresponding to a vector $\mathbf{u}$ is

$$\mathbf{q}_{\mathbf{u}} = (0, \mathbf{u})$$

Other than terminology, we aren't that concerned about Hamilton's intentions for generalized quaternions, because we are only going to consider a specialized case discovered by Arthur Cayley [18]. In particular, he showed that quaternions can be used to describe pure rotations. Later on, Courant and Hilbert [21] determined the relationship between normalized quaternions and the axis–angle representation.

5.5.2 Quaternions as Rotations

While any quaternion can be used to represent rotation (as we will see later), we will be primarily using unit quaternions, where

$$w^2 + \mathbf{v} \cdot \mathbf{v} = 1$$

There are three reasons for this. First of all, it makes the calculations for rotation and conversions more efficient. Secondly, it manages floating-point error. By normalizing, our data will lie in the range $[-1, 1]$, and floating-point values in that range have a high degree of relative precision. Finally, it provides a natural correspondence between an axis–angle rotation and a quaternion. In a unit quaternion, w can be thought of as representing the angle of rotation θ. More specifically, $w = \cos(\theta/2)$. The vector $\mathbf{v}$ represents the axis of rotation, but normalized and scaled by $\sin(\theta/2)$. So, $\mathbf{v} = \sin(\theta/2)\,\hat{\mathbf{r}}$. For example, suppose we wanted to rotate by 90 degrees around the z-axis. Our axis is (0, 0, 1) and half our angle is $\pi/4$ (in radians). The corresponding quaternion components are

$$w = \cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$$

$$x = 0 \cdot \sin\left(\frac{\pi}{4}\right) = 0$$

$$y = 0 \cdot \sin\left(\frac{\pi}{4}\right) = 0$$

$$z = 1 \cdot \sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$$

giving us a final quaternion of

$$\mathbf{q} = \left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right)$$

So, why reformat our previously simple axis and angle to this somewhat strange representation? As we'll see shortly, precooking the data in this way allows us to rotate vectors and concatenate with ease.

Our class implementation for quaternions looks like

```
class IvQuat
{
public:
    // constructor/destructor
    inline IvQuat() {}
    inline IvQuat( float _w, float _x, float _y, float _z )  :
        w(_w), x(_x), y(_y), z(_z)
    {
    }
    IvQuat(const IvVector3& axis, float angle);
    explicit IvQuat(const IvVector3& vector);
    inline ~IvQuat() {}

    // member variables
    float x, y, z, w;
};
```

Much of this follows from what we've already discussed. We can set our quaternion values directly, use an axis–angle format, or explicitly use a vector. Recall that in this last case, we use the vector to set our x, y, and z terms, and set w to 0.

5.5.3 Addition and Scalar Multiplication

Like vectors, quaternions can be scaled and added componentwise. For both operations a quaternion acts just like a 4-vector, so

$$(w_1, x_1, y_1, z_1) + (w_2, x_2, y_2, z_2) = (w_1 + w_2, x_1 + x_2, y_1 + y_2, z_1 + z_2)$$
$$a(w, x, y, z) = (aw, ax, ay, az)$$

The algebraic rules for addition and scalar multiplication that apply to vectors and matrices apply here, so like them, the set of all quaternions is also a vector space. However, the set of unit quaternions is not, since neither operation maintains unit length. Therefore, if we use one of these operations, we'll need to normalize afterwards. In general, however, we will not be using these operations except in special cases.

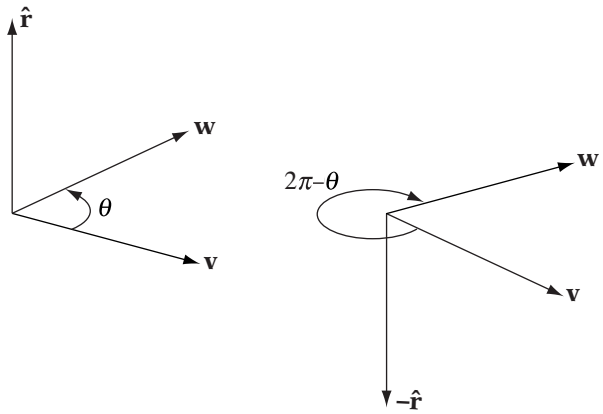


FIGURE 5.6 Comparing rotation performed by a normalized quaternion (left) with its negation (right).

5.5.4 NEGATION

Negation is a subset of scale, but it's worth discussing separately. One would expect that negating a quaternion would produce a quaternion that applies a rotation in the opposite direction—it would be the inverse. However, while it does rotate in the opposite direction, it also rotates around the negative axis. The end result is that a vector rotated by either quaternion ends up in the same place, but if one quaternion rotates by θ radians around $\hat{\mathbf{r}}$, its negation rotates $2\pi - \theta$ radians around $-\hat{\mathbf{r}}$. Figure 5.6 shows what this looks like on the rotation plane. The negated quaternion can be thought of as "taking the other way around," but both quaternions rotate the vector to the same orientation. This will cause some issues when blending between quaternions but can be handled by adjusting our values appropriately, which we'll discuss in Chapter 10. Otherwise, we can use $\mathbf{q}$ and $-\mathbf{q}$ interchangeably.

5.5.5 MAGNITUDE AND NORMALIZATION

As we've implied, we will be normalizing quaternions, and will do so as if we were using 4-vectors. The magnitude of a quaternion is therefore as follows:

$$\|\mathbf{q}\| = \sqrt{(w^2 + x^2 + y^2 + z^2)}$$

A normalized quaternion $\hat{\mathbf{q}}$ is

$$\hat{\mathbf{q}} = \frac{\mathbf{q}}{\|\mathbf{q}\|}$$

Since we're assuming that our quaternions are normalized, we'll forgo the use of the notation $\hat{\mathbf{q}}$ to keep our equations from being too cluttered.

5.5.6 Dot Product

The dot product of two quaternions should also look familiar:

$$\mathbf{q}_1 \cdot \mathbf{q}_2 = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2$$

As with vectors, this is still equal to the cosine of the angle between the quaternions, except that our angle is in four dimensions instead of the usual three. What this gives us is a way of measuring how different two quaternions are. If $\mathbf{q}_1 \cdot \mathbf{q}_2$ is close to 1 (assuming that they're normalized), then they apply very similar rotations. Also, since we know that the negation of a quaternion performs the same rotation as the original, if the dot product is close to -1 the two still apply very similar rotations. So parallel normalized quaternions ($|\mathbf{q}_1 \cdot \mathbf{q}_2| \approx 1$) are similar. Correspondingly, orthogonal normalized quaternions ($\mathbf{q}_1 \cdot \mathbf{q}_2 = 0$) produce extremely different rotations.

5.5.7 Format Conversion

Converting from axis–angle format to a quaternion requires multiplying the angle by one-half, computing the sine and cosine of that result, and scaling the normalized axis vector by the sine. To convert back, we take the arccos of w to get half the angle, and then use $\sqrt{1 - w^2}$ to get the length of $\mathbf{v}$ so we can normalize it. The full conversion is

$$\begin{aligned} \theta &= 2 \arccos(w) \\ \|\mathbf{v}\| &= \sqrt{1 - w^2} \\ \hat{\mathbf{r}} &= \mathbf{v} / \|\mathbf{v}\| \end{aligned}$$

Converting a normalized quaternion to a 3×3 rotation matrix takes the following form:

$$\mathbf{M}_{\mathbf{q}} = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix} \tag{5.6}$$

If the quaternion is not normalized, we need to scale the matrix by

$$\frac{1}{w^2 + x^2 + y^2 + z^2}$$

To compute this on a serial processor we can make use of the fact that there are a lot of duplicated terms. The following is derived from Shoemake [104]:

```
IvMatrix33&
IvMatrix33::Rotation( const IvQuat& q )
{
    float s, xs, ys, zs, wx, wy, wz, xx, xy, xz, yy, yz, zz;

    // if q is normalized, s = 2.0f
    s = 2.0f/( q.x*q.x + q.y*q.y + q.z*q.z + q.w*q.w );

    xs = s*q.x;    ys = s*q.y;    zs = s*q.z;
    wx = q.w*xs;   wy = q.w*ys;   wz = q.w*zs;
    xx = q.x*xs;   xy = q.x*ys;   xz = q.x*zs;
    yy = q.y*ys;   yz = q.y*zs;   zz = q.z*zs;

    mV[0] = 1.0f - (yy + zz);
    mV[3] = xy - wz;
    mV[6] = xz + wy;

    mV[1] = xy + wz;
    mV[4] = 1.0f - (xx + zz);
    mV[7] = yz - wx;

    mV[2] = xz - wy;
    mV[5] = yz + wx;
    mV[8] = 1.0f - (xx + yy);

    return *this;

}   // End of Rotation()
```

If we have a parallel vector processor that can perform fast matrix multiplication, another way of doing this is to generate two 4×4 matrices and multiply them together:

$$\mathbf{M_q} = \begin{bmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{bmatrix} \begin{bmatrix} w & -z & y & -x \\ z & w & -x & -y \\ -y & x & w & -z \\ x & y & z & w \end{bmatrix}$$

If the quaternion is normalized, the product will be the homogeneous rotation matrix corresponding to the quaternion.

To convert a matrix to a quaternion, we can use an approach that is similar to our matrix to axis–angle conversion. Recall that the trace of a rotation matrix is $2\cos\theta + 1$, where θ is our angle of rotation. Also, from equation 5.4, we know that the vector $\mathbf{r} = (R_{21} - R_{12}, R_{02} - R_{20}, R_{10} - R_{01})$ will have length $2\sin\theta$. If we add 1 to the trace and use these as the scalar and vector parts, respectively, of a quaternion, we get

$$\hat{\mathbf{q}} = (2\cos\theta + 2, 2\sin\theta\hat{\mathbf{r}}) \tag{5.7}$$

Surprisingly, all we need to do now is normalize to get the final result. To see why, suppose we started with a quaternion

$$\hat{\mathbf{q}}_1 = (\cos\theta, \sin\theta\hat{\mathbf{r}})$$

This is close to what we need, which is

$$\hat{\mathbf{q}}_h = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\hat{\mathbf{r}})$$

To get from $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_h$, let's consider two vectors. If we have a vector $\mathbf{w}_0$ and a vector $\mathbf{w}_1$ rotated θ degrees from $\mathbf{w}_0$, then to find the vector $\mathbf{v}_h$ that lies between them on the rotation plane (i.e., the vector rotated $\theta/2$ degrees from $\mathbf{w}_0$), we just need to compute $(\mathbf{w}_1 + \mathbf{w}_2)/2$. If we want a normalized vector, we can skip the division by two and just do the normalize step.

So to do the same with quaternions, we take as our $\mathbf{q}_0$ the quaternion $(1, \mathbf{0})$, which represents no rotation. If we add that to $\mathbf{q}_1$ and normalize, that will give us our desired result. That boils down to adding 1 to w and normalizing. Equation 5.7 is just that scaled by 2; the scaling factor drops out nicely when we normalize.

If the trace of the matrix is less than zero, then this will not work. We'll need to use an approach similar to when we extracted the axis from a rotation matrix. By taking the largest diagonal element and subtracting the elements from it, we can derive an equation to solve for a single axis component (e.g., equation 5.5). Using that value as before, we can then compute the other quaternion components from the elements of the matrix.

So, if the largest diagonal element is R_{00}, then

$$x = \frac{1}{2}\sqrt{R_{00} - R_{11} - R_{22} + 1}$$

$$y = \frac{R_{01} + R_{10}}{4x}$$

$$z = \frac{R_{02} + R_{20}}{4x}$$
$$w = \frac{R_{21} - R_{12}}{4x}$$

We can simplify this by noting that

$$4x^2 = R_{00} - R_{11} - R_{22} + 1$$
$$\frac{4x^2}{4x} = \frac{R_{00} - R_{11} - R_{22} + 1}{4x}$$
$$x = \frac{R_{00} - R_{11} - R_{22} + 1}{4x}$$

Substituting this formula for x, we now see that all of the components are scaled by $1/4x$. We can accomplish the same thing by taking the numerators

$$x = R_{00} - R_{11} - R_{22} + 1$$
$$y = R_{01} + R_{10}$$
$$z = R_{02} + R_{20}$$
$$w = R_{21} - R_{12}$$

and normalizing.

Similarly, if the largest diagonal element is R_{11}, we start with

$$y = R_{11} - R_{00} - R_{22} + 1$$
$$x = R_{01} + R_{10}$$
$$z = R_{12} + R_{21}$$
$$w = R_{02} - R_{20}$$

and normalize.

And, if the largest diagonal element is R_{22}, we take

$$z = R_{22} - R_{00} - R_{11} + 1$$
$$x = R_{02} + R_{20}$$
$$y = R_{21} + R_{12}$$
$$w = R_{10} - R_{01}$$

and normalize.

Converting from a fixed-angle format to a quaternion requires creating a quaternion for each rotation around a coordinate axis, and then concatenating them together. For the *z*-*y*-*x* fixed-angle format, the result is

$$
\begin{aligned}
w &= \cos\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\cos\frac{\theta_z}{2} - \sin\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\sin\frac{\theta_z}{2} \\
x &= \sin\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\cos\frac{\theta_z}{2} + \cos\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\sin\frac{\theta_z}{2} \\
y &= \cos\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\cos\frac{\theta_z}{2} - \sin\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\sin\frac{\theta_z}{2} \\
z &= \cos\frac{\theta_x}{2}\cos\frac{\theta_y}{2}\sin\frac{\theta_z}{2} + \sin\frac{\theta_x}{2}\sin\frac{\theta_y}{2}\cos\frac{\theta_z}{2}
\end{aligned}
$$

Converting a quaternion to fixed or Euler angles is, quite frankly, an awful thing to do. If it's truly necessary (e.g., for an interface), the simplest method is to convert the quaternion to a matrix, and extract the Euler angles from the matrix.

5.5.8 Concatenation

As with matrices, if we wish to concatenate the transformations performed by two quaternions, we multiply them together to get a new quaternion. Expanding out the terms of the multiplication produces the following result:

$$
\begin{aligned}
&(w_2 + x_2\boldsymbol{i} + y_2\boldsymbol{j} + z_2\boldsymbol{k})(w_1 + x_1\boldsymbol{i} + y_1\boldsymbol{j} + z_1\boldsymbol{k}) \\
&\quad = w_2w_1 + w_2x_1\boldsymbol{i} + w_2y_1\boldsymbol{j} + w_2z_1\boldsymbol{k} \\
&\qquad + x_2w_1\boldsymbol{i} + x_2x_1\boldsymbol{i}^2 + x_2y_1\boldsymbol{ij} + x_2z_1\boldsymbol{ik} \\
&\qquad + y_2w_1\boldsymbol{j} + y_2x_1\boldsymbol{ji} + y_2y_1\boldsymbol{j}^2 + y_2z_1\boldsymbol{jk} \\
&\qquad + z_2w_1\boldsymbol{k} + z_2x_1\boldsymbol{ki} + z_2y_1\boldsymbol{kj} + z_2z_1\boldsymbol{k}^2
\end{aligned}
\tag{5.8}
$$

We define the products of the $\boldsymbol{i}$, $\boldsymbol{j}$, and $\boldsymbol{k}$ quantities as follows:

$$
\begin{aligned}
\boldsymbol{ij} &= \boldsymbol{k} \quad & \boldsymbol{jk} &= \boldsymbol{i} \quad & \boldsymbol{ki} &= \boldsymbol{j} \\
\boldsymbol{ji} &= -\boldsymbol{k} \quad & \boldsymbol{kj} &= -\boldsymbol{i} \quad & \boldsymbol{ik} &= -\boldsymbol{j}
\end{aligned}
$$

and

$$
\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1
$$

Note that order does matter.

We can use these properties and well-known vector operations to simplify the product to

$$\mathbf{q}_2 \cdot \mathbf{q}_1 = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_2 \times \mathbf{v}_1)$$

Note that we've expressed this in a right-to-left order, like our matrices. This is because the rotation defined by $\mathbf{q}_1$ will be applied first, followed by the rotation defined by $\mathbf{q}_2$. We'll see this more clearly when we look at how we use quaternions to transform vectors. Also note the cross product; due to this, quaternion multiplication is also not commutative. This is what we expect with rotations; applying two rotations in one order does not necessarily provide the same result as applying them in the reverse order.

Multiplying two normalized quaternions does produce a normalized quaternion. However, due to floating-point error, it is wise to renormalize the result—if not after every multiplication, at least often and definitely before using the quaternion to rotate vectors.

A straightforward implementation of quaternion multiplication might look like

```
IvQuat operator*(IvQuat q2, IvQuat q1)
{
    IvVector3 v1(q1.x, q1.y, q1.z);
    IvVector3 v2(q2.x, q2.y, q2.z);

    float w = q1.w*q2.w - v1.Dot(v2);
    IvVector3 v = q1.w*v2 + q2.w*v1 + v2.Cross(v1);
    IvQuat q(w, v);

    return q;
}
```

Alternatively, we can unroll the operations to get

```
IvQuat operator*(IvQuat q2, IvQuat q1)
{
    w = q2.w*q1.w - q2.x*q1.x
        - q2.y*q1.y - q2.z*q1.z;
    x = q2.y* q1.z - q2.z*q1.y
        + q2.w*q1.x + q1.w*q2.x;
    y = q2.z*q1.x - q2.x*q1.z
        + q2.w*q1.y + q1.w*q2.y;
```

```
    z = q2.x*q1.y - q2.y*q1.x
        + q2.w*q1.z + q1.w*q2.z;
    return IvQuat(w,x,y,z);
}
```

Note that on a scalar processor that concatenating two quaternions can actually be faster than multiplying two matrices together.

An example of concatenating quaternions is the conversion from *z*-*y*-*x* fixed-angle format to a quaternion. The corresponding quaternions for each axis are

$$\mathbf{q}_z = \left(\cos\frac{\theta_z}{2}, 0, 0, \sin\frac{\theta_z}{2}\right)$$

$$\mathbf{q}_y = \left(\cos\frac{\theta_y}{2}, 0, \sin\frac{\theta_y}{2}, 0\right)$$

$$\mathbf{q}_x = \left(\cos\frac{\theta_x}{2}, \sin\frac{\theta_x}{2}, 0, 0\right)$$

Multiplying these together in the order $\mathbf{q}_x\mathbf{q}_y\mathbf{q}_z$ gives the result in Section 5.5.7.

5.5.9 Identity and Inverse

As with matrix products, there is an identity quaternion and, subsequently, there are multiplicative inverses. As we've mentioned, the identity quaternion is (1, 0, 0, 0), or (1, $\mathbf{0}$). Multiplying this by any quaternion $\mathbf{q} = (w, \mathbf{v})$ gives

$$\begin{aligned}\mathbf{q}\cdot(1,\mathbf{0}) &= (1\cdot w - \mathbf{0}\cdot\mathbf{v}, 1\mathbf{v} + w\mathbf{0} + \mathbf{v}\times\mathbf{0})\\ &= (w, \mathbf{v})\end{aligned}$$

In this case, multiplication is commutative, so $\mathbf{q}\cdot(1,\mathbf{0}) = (1,\mathbf{0})\cdot\mathbf{q} = \mathbf{q}$.

As with matrices, the inverse $\mathbf{q}^{-1}$ of a quaternion $\mathbf{q}$ is one such that $\mathbf{q}^{-1}\mathbf{q} = \mathbf{q}\mathbf{q}^{-1} = (1,\mathbf{0})$. If we consider a quaternion as rotating θ degrees counterclockwise around an axis $\hat{\mathbf{r}}$, then to undo the rotation we should rotate θ degrees clockwise around the same axis. This is the same as rotating $-\theta$ degrees counterclockwise: To create the inverse we negate the angle (Figure 5.7(a)). So, if

$$(w, \mathbf{v}) = \left(\cos\left(\frac{\theta}{2}\right), \hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right)\right)$$

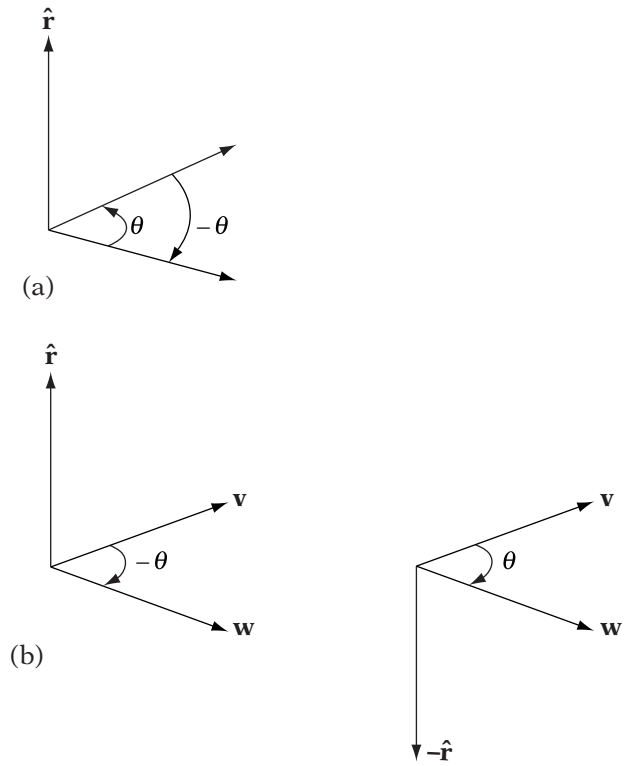


FIGURE 5.7 (a) Relationship between quaternion and its inverse. Inverse rotates around the same axis but negative angle. (b) Rotation direction around axis by negative angle is the same as rotation direction around negative axis by positive angle.

then

$$\begin{aligned}(w, \mathbf{v})^{-1} &= \left(\cos\left(-\frac{\theta}{2}\right), \hat{\mathbf{r}}\sin\left(-\frac{\theta}{2}\right)\right) \\ &= \left(\cos\left(\frac{\theta}{2}\right), -\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right)\right) \\ (w, \mathbf{v})^{-1} &= (w, -\mathbf{v})\end{aligned} \tag{5.9}$$

At first glance, negating the vector part of the quaternion (also known as the *conjugate*) to reverse the rotation is counterintuitive. But after some thought this still makes sense geometrically. A clockwise rotation around an axis turns in the same direction as a counterclockwise rotation around the negative of the axis (Figure 5.7(b)).

Equation 5.9 only holds if our quaternion is normalized. While in most cases it should be since we're trying to maintain unit quaternions, if it is not then we need to scale by one over the length squared, or

$$\mathbf{q}^{-1} = \frac{1}{\|\mathbf{q}\|^2}(w, -\mathbf{v}) \tag{5.10}$$

Avoiding the floating-point divide in this case is another good reason to keep our quaternions normalized.

Equation 5.10 may make more sense if we consider the inverse of a quaternion $s\hat{\mathbf{q}}$ (i.e., a nonunit quaternion with magnitude s):

$$\begin{aligned}(s\hat{\mathbf{q}})^{-1} &= (s(w, \mathbf{v}))^{-1} \\ &= \frac{1}{s^2}s(w, -\mathbf{v}) \\ &= \frac{1}{s}(w, -\mathbf{v}) \\ &= \frac{1}{s}\hat{\mathbf{q}}^{-1}\end{aligned}$$

It bears repeating that the negative of a quaternion, where both w and $\mathbf{v}$ are negated, is not the same as the inverse. When applied to vectors, the negative actually rotates the vector to the same orientation but going the other way around the axis.

5.5.10 Vector Rotation

If $\mathbf{qr}$ is used to concatenate two quaternions $\mathbf{q}$ and $\mathbf{r}$, then for a vector $\mathbf{p}$ we might expect $\mathbf{qp}$ to rotate the vector by the quaternion, just as it does for a matrix. Unfortunately for intuition, this is not the case. For one thing, the result of this multiplication is not a vector (w will not be 0). The actual formula for rotating a vector by a quaternion is

$$R_{\mathbf{q}}\mathbf{p} = \mathbf{qpq}^{-1} \tag{5.11}$$

It may look like the effect of the operation is to perform the rotation and then undo it, but this is not the case. Remember that quaternion multiplication is not commutative, so if $\mathbf{q}$ is not the identity:

$$\mathbf{qpq}^{-1} \neq \mathbf{qq}^{-1}\mathbf{p} = \mathbf{p}$$

We can use our rotation formula for axis and angle to show that equation 5.11 does rotate a vector. We begin by breaking it out into its

component vector operations. Assuming that our quaternion is normalized, if we expand the full multiplication and combine terms, we get

$$R_{\mathbf{q}}\mathbf{p} = (2w^2 - 1)\mathbf{p} + 2(\mathbf{v}\cdot\mathbf{p})\mathbf{v} + 2w(\mathbf{v}\times\mathbf{p}) \tag{5.12}$$

Substituting $\cos(\theta/2)$ for w, and $\hat{\mathbf{r}}\sin(\theta/2)$ for $\mathbf{v}$, we get

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) = &\left(2\cos^2\left(\frac{\theta}{2}\right) - 1\right)\mathbf{p} + \left(\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right)\cdot\mathbf{p}\right)\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right) \\ &+ 2\cos\left(\frac{\theta}{2}\right)\left(\hat{\mathbf{r}}\sin\left(\frac{\theta}{2}\right)\times\mathbf{p}\right) \end{aligned}$$

Reducing terms and using the appropriate trigonometric identities, we end up with

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= \left(\cos^2\left(\frac{\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right)\right)\mathbf{p} + 2\sin^2\left(\frac{\theta}{2}\right)(\hat{\mathbf{r}}\cdot\mathbf{p})\hat{\mathbf{r}} \\ &\quad + 2\cos\left(\frac{\theta}{2}\right)\sin\left(\frac{\theta}{2}\right)(\hat{\mathbf{r}}\times\mathbf{p}) \\ &= \cos\theta\mathbf{p} + [1-\cos\theta](\hat{\mathbf{r}}\cdot\mathbf{p})\hat{\mathbf{r}} + \sin\theta(\hat{\mathbf{r}}\times\mathbf{p}) \end{aligned} \tag{5.13}$$

We see that equation 4.13 is equal to equation 5.13, so our quaternion multiplication — odd as it may look — does rotate a vector around an axis by a given angle.

In our code, we won't want to use the $\mathbf{qpq}^{-1}$ form, since performing both quaternion multiplications isn't very efficient. Instead, we'll use equation 5.12:

```
IvVector3
IvQuat::Rotate( const IvVector3& vector ) const
{
    ASSERT( IsUnit() );

    float vMult = 2.0f*(x*vector.x + y*vector.y + z*vector.z);
    float crossMult = 2.0f*w;
    float pMult = crossMult*w - 1.0f;

    return IvVector3( pMult*vector.x + vMult*x + crossMult*(y*vector.z - z*vector.y),
                      pMult*vector.y + vMult*y + crossMult*(z*vector.x - x*vector.z),
                      pMult*vector.z + vMult*z + crossMult*(x*vector.y - y*vector.x) );

}   // End of IvQuat::Rotate()
```

The operation count is more than that of matrix multiplication, but comparable to Rodrigues' formula for axis–angle representation.

An alternate version,

$$R_{\mathbf{q}}\mathbf{p} = (\mathbf{v}\cdot\mathbf{p})\mathbf{v} + w^2\mathbf{p} + 2w(\mathbf{v}\times\mathbf{p}) + \mathbf{v}\times(\mathbf{v}\times\mathbf{p})$$

is useful for processors that have fast cross product operations.

Neither of these formulas is as efficient as matrix multiplication, but for a single vector it is more efficient to perform these operations rather than convert the quaternion to a matrix and then multiply. However, if we need to rotate multiple vectors by the same quaternion, matrix conversion becomes worthwhile.

To see how concatenation of rotations works, suppose we apply a rotation from one quaternion followed by a second rotation from another quaternion. We can rearrange parentheses to get

$$\mathbf{q}(\mathbf{r}\mathbf{p}\mathbf{r}^{-1})\mathbf{q}^{-1} = (\mathbf{q}\mathbf{r})\mathbf{p}(\mathbf{q}\mathbf{r})^{-1}$$

As we see, concatenated quaternions will apply their rotation, one after the other. The order is right to left, as we have stated.

If we substitute $-\mathbf{q}$ in place of $\mathbf{q}$ in equation 5.11, we can see in another way how negating the quaternion doesn't affect rotation. By equation 5.10, $(-\mathbf{q})^{-1} = -\mathbf{q}^{-1}$, so

$$\begin{aligned} R_{-\mathbf{q}}(\mathbf{p}) &= -\mathbf{q}\mathbf{p}(-\mathbf{q})^{-1} \\ &= \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \end{aligned}$$

The two negatives cancel, and we're back with our familiar result.

Similarly, if $\mathbf{q}$ is a nonunit quaternion, we can show that the same result occurs as if the quaternion were normalized:

$$\begin{aligned} (s\hat{\mathbf{q}})\mathbf{p}(s\hat{\mathbf{q}})^{-1} &= (s\hat{\mathbf{q}})\mathbf{p}(\frac{1}{s}\hat{\mathbf{q}}^{-1}) \\ &= s\frac{1}{s}\hat{\mathbf{q}}\mathbf{p}\hat{\mathbf{q}}^{-1} \\ &= \hat{\mathbf{q}}\mathbf{p}\hat{\mathbf{q}}^{-1} \end{aligned}$$

5.5.11 Shortest Path of Rotation

As with the axis–angle format, it is often useful to create a quaternion that rotates a vector $\mathbf{v}_1$ into another vector $\mathbf{v}_2$, although in this case we'll use a different approach discussed by Baker and Norel [5] that also avoids some issues with numerical error when $\mathbf{v}_1$ and $\mathbf{v}_2$ are nearly collinear.

We begin by taking the dot product and cross product of the two vectors:

$$\mathbf{v}_1 \cdot \mathbf{v}_2 = \|\mathbf{v}_1\|\|\mathbf{v}_2\| \cos\theta$$
$$\mathbf{v}_1 \times \mathbf{v}_2 = \|\mathbf{v}_1\|\|\mathbf{v}_2\| \sin\theta\hat{\mathbf{r}}$$

where $\hat{\mathbf{r}}$ is our normalized rotation axis. Using these as the scalar and vector parts, respectively, of a quaternion and normalizing gives us

$$\hat{\mathbf{q}}_1 = (\cos\theta, \sin\theta\hat{\mathbf{r}})$$

This should look familiar from our previous discussion of matrix to quaternion conversion. As before, if we add 1 to w,

$$\hat{\mathbf{q}}_h = (\cos\theta + 1, \sin\theta\hat{\mathbf{r}})$$

and normalize, we get

$$\hat{\mathbf{q}} = (\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\hat{\mathbf{r}})$$

Note that we haven't handled the case where the two vectors are parallel. In this case, there are an infinite number of possible rotation axes, and hence an infinite number of possible quaternions. A stop-gap solution is to pick one by taking the cross product between one of the vectors and a known vector such as **i** or **j**. While this will work, it may lead to discontinuities — something we'll discuss in Chapter 10 when we cover interpolation.

5.5.12 Quaternions and Transformations

While quaternions are good for rotations, they don't help us much when performing translation and scale. Fortunately, we already have a transformation format that quaternions fit right into. Recall that in Chapter 4, instead of using a generalized 4×4 matrix for affine transformations, we used a single scale factor s, a 3×3 rotation matrix $\mathbf{R}$, and a translation vector $\mathbf{t}$. Our formula for transformation was

$$\mathbf{p}' = \mathbf{R}(s\mathbf{p}) + \mathbf{t}$$

We can easily replace our matrix $\mathbf{R}$ with an equivalent quaternion $\mathbf{r}$, which gives us

$$\mathbf{p}' = \mathbf{r}(s\mathbf{p})\mathbf{r}^{-1} + \mathbf{t}$$

Concatenation using the quaternion is similar to concatenation with our original separated format, except that we replace multiplication by the rotation matrix with quaternion operations:

$$s' = s_1 s_0$$
$$\mathbf{r}' = \mathbf{r}_1 \mathbf{r}_0$$
$$\mathbf{t}' = \mathbf{t}_1 + \mathbf{r}_1 (s_1 \mathbf{t}_0) \mathbf{r}_1^{-1}$$

Again, to add the translations, we first need to scale $\mathbf{t}_0$ by s_1 and then rotate by the quaternion $\mathbf{r}_1$.

As with lone quaternions, concatenation on a serial processor can be much cheaper in this format than using a 4×4 matrix. However, transformation of points is more expensive. As was the case with simple rotation, for multiple points it will be better to convert the quaternion to a matrix and transform them that way.

5.6 Chapter Summary

In this chapter we've discussed four different representations for orientation and rotation: matrices, fixed and Euler angles, axis and angle, and quaternions. In the introduction we gave three criteria for our format: It may be informative to compare them along with their usefulness in interpolation.

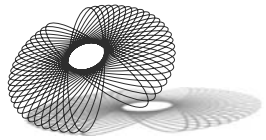
As far as size, matrices are the worst at nine values, and fixed and Euler angles are the best at three values. However, quaternions and axis–angle representation are close to fixed and Euler angles at four values, and they avoid the problems engendered by gimbal lock.

For concatenation, quaternions take the fewest number of operations, followed closely by matrices, and then by axis–angle and fixed and Euler representations. The last two are hampered by not having low-cost methods for direct concatenation and so the majority of their expense is tied up in converting to a more favorable format.

When transforming vectors, matrices are the clear winner. Assuming precached sine and cosine data, fixed and Euler angles are close behind, while axis–angle representation and quaternions take a bit longer. However, if we don't precache our data, the sine and cosine computations will probably take longer, and quaternions come in second.

Finally, it is worth noting that due to floating-point error, the numbers representing our orientation may drift. The axis–angle and fixed and Euler angle formats do not provide an intuitive method for correcting for this. On the other hand, matrices can use Gram-Schmidt orthonormalization and quaternions can perform a normalization step. Quaternions are a clear winner here as normalizing four values is a relatively inexpensive operation.

For further reading about quaternions, the best place to start is with the writings of Shoemake, in particular [103]. Hamilton's original series of articles on quaternions [52] are in the public domain and can be found by searching online. Courant and Hilbert [21] cover applications of quaternions, in particular to represent rotations. Finally, Eberly has an article [26] comparing orientation formats, and an entire chapter in his latest book [27] on quaternions, with additional material by Shoemake.

Chapter 6

Viewing and Projection

6.1 Introduction

In previous chapters we've discussed how to represent objects, basic transformations we can apply to these objects, and how we can use these transformations to move and manipulate our objects within our virtual world. With that background in place, we can begin to discuss the mathematics underlying the techniques we use to display our game objects on a monitor or other visual display medium.

It doesn't take much justification to understand why we might want to view the game world — after all, games are primarily a visual media. Other sensory outputs are of course possible, particularly sound and haptic (or touch) feedback. Both have become more sophisticated and in their own way provide another representation of the relative three-dimensional (3D) position and orientation of game objects. But in the current market, when we think of games, we first think of what we can see.

To achieve this, we'll be using a continuation of our transformation process known as the *graphics pipeline*. Figure 6.1 shows the situation. We already have a transformation that takes our model from its local space to world space. At each stage of the graphics pipeline, we continue to concatenate matrices to this matrix. Our goal is to build a single matrix to transform the points in our object from their local configuration to a two-dimensional (2D) representation suitable for displaying.

The first part of the display process involves setting up a virtual viewer or camera, which allows us to control which objects lie in our current view. As we'll see, this camera is just like any other object in the game; we can

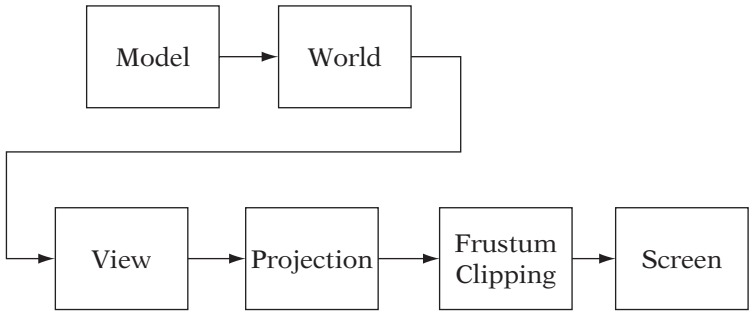


Figure 6.1 The graphics pipeline.

set the camera's position and orientation based on an affine transformation. Inverting this transformation is the first stage of our pipeline: It allows us to transform objects in the world frame into the point of view of the camera object.

From there we will want to build and concatenate a matrix that transforms our objects in view into coordinates so they can be represented in an image. This flattening or *projection* takes many forms, and we'll discuss several of the most commonly used projections. In particular, we'll derive perspective projection, which most closely mimics our viewpoint of the real world.

At this point, it is usually convenient to cull out any objects that will not be visible on our screen, and possibly cut, or clip, others that intersect the screen boundaries. This will make our final rendering process much faster.

The final stage is to transform our projected coordinates and stretch and translate them to fit a specific portion of the screen, known as the viewport. This is known as the screen transformation.

In addition, we'll cover how to reverse this process so we can take a mouse click on our 2D screen and use it to select objects in our 3D world. This process, known as *picking*, can be useful when building an interface with 3D elements. For example, selecting units in a 3D real-time strategy game is done via picking.

As with other chapters, we'll be discussing how to implement these transformations in production code. Because our primary platform is OpenGL, for the most part we'll be focusing on its pipeline and how it handles the viewing and projective transformations. However, we will also cover the cases where it may differ from graphics APIs, particularly Direct3D.

One final note before we begin: There is no standard representation for this process. In other books you may find these stages broken up in different ways, depending on the rendering system the authors are trying to present. However, the ultimate goal is the same: Take an object in the world and transform it from a viewer's perspective onto a 2D medium.

6.2 View Frame and View Transformation

6.2.1 Defining a Virtual Camera

In order to render objects in the world, we need to represent the notion of a viewer. This could be the main character's viewpoint in a first-person shooter, or an over-the-shoulder view in a third-person adventure game, or a zoomed-out wide shot in a strategy game. We may want to control properties of our viewer to simulate a virtual camera, for example, we may want to create an in-game scripted sequence where we pan across a screen or follow a set path through a space. We encapsulate these properties into a single entity, commonly called the *camera*.

For now, we'll consider only the most basic properties of the camera needed for rendering. We are trying to answer two questions [8]: Where am I? Where am I looking? We can think of this as someone taking an actual camera, placing it on a tripod, and aiming it at an object of interest.

The answer to the first question is the camera's position, E, which is variously called the *eyepoint*, the *view position*, or the *view space origin*. As we mentioned, this could be the main character's eye position, a location over his shoulder, or a spot pulled back from the action. While this position can be placed relative to another object's location, it is usually cleaner and easier to manage if we represent it in the world frame.

A partial answer to the second question is a vector called the *view direction vector*, or $\mathbf{v}_{dir}$, which points along the facing direction for the camera. This could be a vector from the camera position to an object or point of interest, a vector indicating the direction the main character is facing, or a fixed direction if we're trying to simulate a top-down view for a strategy game. For the purposes of setting up the camera, this is also specified in the world frame.

Having a single view direction vector is not enough to specify our orientation, since there are an infinite number of rotations around that vector. To constrain our possibilities down to one, we specify a second vector orthogonal to the first, called the *view up vector*, or $\mathbf{v}_{up}$. This indicates the direction out of the top of the camera. From these two we can take the cross product to get the *view side vector*, or $\mathbf{v}_{side}$, which usually points out toward the camera's right. Normalizing these three vectors and adding the view position gives us an orthonormal basis and an origin, or an affine frame. This is the camera's local frame, also known as the *view frame*, (Figure 6.2).

The three view vectors specify where the view orientation is relative to the world frame. However, we also need to define where these vectors are from the perspective of the camera. The standard order used by most viewing systems is to make the camera's y-axis represent the view up vector in the camera's local space, and the camera's x-axis represent the corresponding view side vector. This aligns our camera's local coordinates so that x values vary left

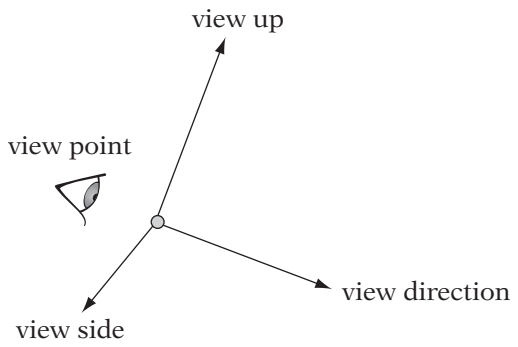


Figure 6.2 View frame relative to the world frame.

and right along the plane of the screen and y values vary up and down, which is very intuitive.

The remaining question is what to do with z and the view direction. In most systems, the z-axis is treated as the camera-relative view direction vector (Figure 6.3(a)). This has a nice intuitive feel: As objects in front of the viewer move farther away, their z values relative to the camera will increase. The value of z can act as a measure of the distance between the object and the camera, which we can use for hidden object removal. Note, however, that this is a left-handed system, as $(\hat{\mathbf{v}}_{side} \times \hat{\mathbf{v}}_{up}) \cdot \hat{\mathbf{v}}_{dir} < 0$.

OpenGL does not follow the standard model; instead, it chooses a slightly different approach. It maintains a right-handed system where the camera-relative view direction is aligned with the negative z-axis (Figure 6.3(b)). So in this case, the farther away the object is, its $-z$ coordinate gets larger relative to the camera. This is not as convenient for distance calculations, but it does allow us to remain in a right-handed coordinate system. This avoids having to worry about reflections when transforming from the world frame to the view frame, as we'll see below.

6.2.2 Constructing the View-to-World Transformation

Now that we have a way of representing and setting camera position and orientation, what do we do with it? The first step in the rendering process is to move all of the objects in our world so that they are no longer relative to the world frame, but are relative to the camera's view. Essentially, we want to transform the objects from the world frame to the view frame. This gives us a sense of what we can see from our camera position. In the view frame, those objects along the line of the view direction vector (i.e., the $-z$-axis in the case of OpenGL) are in front of the camera and so will most

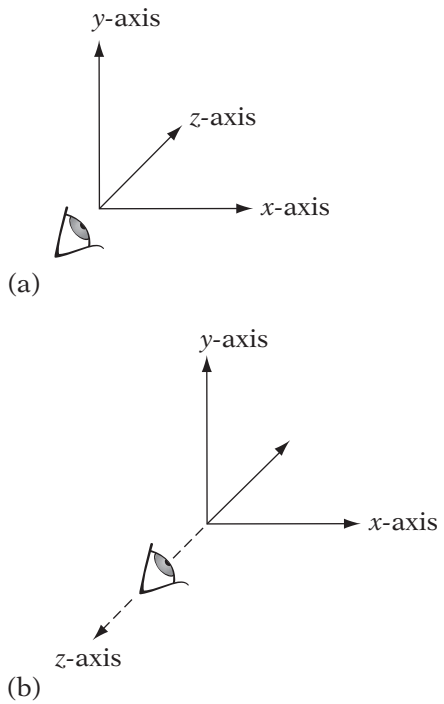


FIGURE 6.3 (a) Standard view frame axes. (b) OpenGL view frame axes.

likely be visible in our scene. Those on the other side of the plane formed by the view position, the view side vector, and the view up vector are behind the camera, and therefore not visible. In order to achieve this situation, we need to create a transformation from world space to view space, known as the world-to-view transformation, or more simply, the view transformation. We can represent this transformation as $\mathbf{M}_{world \to view}$.

However, rather than building this transformation directly, we usually find it easier to build $\mathbf{M}^{-1}_{world \to view}$, or $\mathbf{M}_{view \to world}$, first, and then invert to get our final world-to-view frame transformation. In order to build this, we'll make use of the principles we introduced in Chapter 4. If we look again at Figure 6.2, we note that we have an affine frame — the view frame — represented in terms of the world frame.

We can use this information to define the transformation from the view frame to the world frame as a 4×4 affine matrix. The origin E of the view frame is translated to the view position, so the translation vector $\mathbf{y}$ is equal to $E - O$. We'll abbreviate this as $\mathbf{v}_{pos}$. Similarly, the view vectors represent how the standard basis vectors in view space are transformed into world space and become columns in the upper left 3×3 matrix $\mathbf{A}$. To build $\mathbf{A}$, however, we need

to define which standard basis vector in the view frame maps to a particular view vector in the world frame.

Recall that in the standard case, the camera's local x-axis represents $\hat{\mathbf{v}}_{side}$, the y-axis represents $\hat{\mathbf{v}}_{up}$, and the z-axis represents $\hat{\mathbf{v}}_{dir}$. This mapping indicates which columns the view vectors should be placed in, and the view position translation vector takes its familiar place in the right-most column. The corresponding transformation matrix is

$$\mathbf{A} = \begin{bmatrix} \hat{\mathbf{v}}_{side} & \hat{\mathbf{v}}_{up} & \hat{\mathbf{v}}_{dir} & \mathbf{v}_{pos} \end{bmatrix} \tag{6.1}$$

Note that in this case we are mapping from a left-handed view frame to the right-handed world frame, so the upper 3×3 is not a pure rotation but a rotation concatenated with a reflection.

For OpenGL, the only change is that we want to look down the $-z$-axis. This is the same as the z-axis mapping to the negative view direction vector. So, the corresponding matrix is

$$\mathbf{A} = \begin{bmatrix} \hat{\mathbf{v}}_{side} & \hat{\mathbf{v}}_{up} & -\hat{\mathbf{v}}_{dir} & \mathbf{v}_{pos} \end{bmatrix} \tag{6.2}$$

In this case, since we are mapping from a right-handed frame to a right-handed frame, no reflection is necessary, and the upper 3×3 matrix is a pure rotation. Not having a reflection can actually be a benefit, particularly with some culling methods.

6.2.3 Controlling the Camera

It's not enough that we have a transformation for our camera that encapsulates position and orientation. More often we'll want to move it around the world. Positioning our camera is a simple enough matter of translating the view position, but controlling view orientation is another problem. One way is to specify the view vectors directly and build the matrix as described. This assumes, of course, that we already have a set of orthogonal vectors we want to use for our viewing system.

The more usual case is that we only know the view direction. For example, suppose we want to continually focus on a particular object in the world (known as the look-at object). We can construct the view direction by subtracting the view position from the object's position. But whether we have a given view direction or we generate it from the look-at object, we still need two other orthogonal vectors to properly construct an orthogonal basis. We can calculate them by using one additional piece of information: the world up vector. This is a fixed vector representing the "up" direction in the world frame. In our case, we'll use the z-axis basis vector $\mathbf{k}$ (Figure 6.4), although in general, any vector that we care to call "up" will do. For example, suppose we

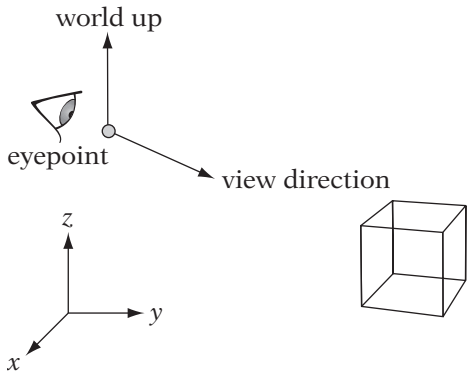


FIGURE 6.4 Look-at representation.

had a mission on a boat at sea and wanted to give the impression that the boat was rolling from side to side, without affecting the simulation. One method is to change the world up vector over time, oscillating between two keeled-over orientations, and use that to calculate your camera orientation.

For now, however, we'll use $\mathbf{k}$ as our world up vector. Our goal is to compute orthonormal vectors in the world frame corresponding to our view vectors, such that one of them is our view direction vector $\hat{\mathbf{v}}_{dir}$, and our view up vector $\hat{\mathbf{v}}_{up}$ matches the world up vector as closely as possible. Recall that we can use Gram-Schmidt orthogonalization to create orthogonal vectors from a set of nonorthogonal vectors, so

$$\mathbf{v}_{up} = \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir})\,\hat{\mathbf{v}}_{dir}$$

Normalizing gives us $\hat{\mathbf{v}}_{up}$. We can take the cross product to get the view side vector:

$$\hat{\mathbf{v}}_{side} = \hat{\mathbf{v}}_{dir} \times \hat{\mathbf{v}}_{up}$$

We don't need to normalize in this case because the two vector arguments are orthonormal. The resulting vectors can be placed as columns in the transformation matrix as before.

One problem may arise if we are not careful: What if $\hat{\mathbf{v}}_{dir}$ and $\mathbf{k}$ are parallel? If they are equal, we end up with

$$\begin{aligned}\mathbf{v}_{up} &= \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir})\,\hat{\mathbf{v}}_{dir} \\ &= \mathbf{k} - 1 \cdot \hat{\mathbf{v}}_{dir} \\ &= \mathbf{0}\end{aligned}$$

If they point in opposite directions we get

$$\begin{aligned}\mathbf{v}_{up} &= \mathbf{k} - (\mathbf{k} \cdot \hat{\mathbf{v}}_{dir})\,\hat{\mathbf{v}}_{dir} \\ &= \mathbf{k} - (-1) \cdot \hat{\mathbf{v}}_{dir} \\ &= \mathbf{0}\end{aligned}$$

Clearly, neither case will lead to an orthonormal basis.

The recovery procedure is to pick an alternative vector that we know is not parallel, such as **i** or **j**. This will lead to what seems like an instantaneous rotation around the z-axis. To understand this, raise your head upward until you are looking at the ceiling. If you keep going, you'll end up looking at the wall behind you, but upside down. To maintain the view looking right-side up, you'd have to rotate your head 180 degrees around your view direction (don't try this at home). This is not a very pleasing result, so avoid aligning the view direction with the world up vector whenever possible.

There is a third possibility for controlling camera orientation. Suppose we want to treat our camera just like a normal object and specify a rotation matrix and translation vector. To do this we'll need to specify a starting orientation Ω for our camera and then apply our rotation matrix to find our camera's final orientation, after which we can apply our translation. Which orientation is chosen is somewhat arbitrary, but some are more intuitive and convenient than others. In our case, we'll say that in our default orientation the camera has an initial view direction along the world x-axis, an initial view up along the world z-axis, and an initial view side along the $-y$-axis. This aligns the view up vector with the world up vector, and using the x-axis as the view direction fits the convention we set for objects' local space in Chapter 4.

Substituting these values into the view-to-world matrix for the standard left-handed view frame (equation 6.1) gives

$$\mathbf{\Omega}_s = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The equivalent matrix for the right-handed OpenGL view frame (using equation 6.2) is

$$\mathbf{\Omega}_{ogl} = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Whichever system we are using, after this we apply our rotation to orient our frame in the direction we wish and, finally, the translation for the view

position. If the three column vectors in our rotation matrix are **u**, **v**, and **w**, then for OpenGL the final transformation matrix is

$$\begin{aligned}\mathbf{M}_{view\rightarrow world} &= \mathbf{TR\Omega}_{ogl} \\ &= \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -\mathbf{j} & \mathbf{k} & -\mathbf{i} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} -\mathbf{v} & \mathbf{w} & -\mathbf{u} & \mathbf{v}_{pos} \\ 0 & 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

6.2.4 Constructing the World-to-View Transformation

Using the techniques in the previous two sections, now we can create a transformation that takes us from view space to world space. To create the reverse operator, we need only to invert the transformation. Since we know that it is an affine transformation, we can invert it as

$$\mathbf{M}_{world\rightarrow view} = \begin{bmatrix} \mathbf{R}^{-1} & -(\mathbf{R}^{-1}\mathbf{v}_{pos}) \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where **R** is the upper 3×3 block of our view-to-world transformation. And since **R** is the product of either a reflection and rotation matrix (in the standard case) or two rotations (in the OpenGL case), it is an orthogonal matrix, so we can compute its inverse by taking the transpose:

$$\mathbf{M}_{world\rightarrow view} = \begin{bmatrix} \mathbf{R}^{T} & -(\mathbf{R}^{T}\mathbf{v}_{pos}) \\ \mathbf{0}^T & 1 \end{bmatrix}$$

In practice, this transformation is usually calculated directly, rather than taking the inverse of an existing transformation. For example, OpenGL has a utility call `gluLookAt()` that computes the view transformation assuming a view position, desired view position, and world up vector. One possible implementation is as follows.

```
void LookAt( const IvVector3& eye,
             const IvVector3& lookAt,
             const IvVector3& up )
{
  // compute view vectors
  IvVector3 viewDir = lookAt - eye;
  IvVector3 viewSide;
```

```
    IvVector3 viewUp;
    viewDir.Normalize();
    viewUp = up - up.Dot(viewDir)*viewDir;
    viewUp.Normalize();
    viewSide = viewDir.Cross(viewUp);

    // now set up matrices
    // build transposed rotation matrix
    IvMatrix33 rotate;
    rotate.SetRows( viewSide, viewUp, -viewDir );

    // transform translation

    IvVector3 eyeInv = -(rotate*eye);

    // build 4x4 matrix
    IvMatrix44 matrix;
    matrix.Rotation(rotate);
    matrix(0,3) = eyeInv.x;
    matrix(1,3) = eyeInv.y;
    matrix(2,3) = eyeInv.z;

    // set view to world transformation
    ::SetViewTransform( matrix.mV );
}
```

Note that we use the method `IvMatrix33:SetRows()` to set the transformed basis vectors since we're setting up the inverse matrix, namely, the transpose. There is also no recovery code if the view direction and world up vectors are collinear—it is assumed that any external routine will ensure this does not happen. The renderer method `::SetViewTransform()` stores the calculated view transformation and is discussed in more detail in Section 6.7.

6.3 Projective Transformation

6.3.1 Definition

Now that we have a method for controlling our view position and orientation, and for transforming our objects into the view frame, we can look at the second stage of the graphics pipeline: taking our 3D space and transforming it into a form suitable for display on a 2D medium. This process of transforming from $\mathbb{R}^3$ to $\mathbb{R}^2$ is called *projection*.

We've already seen one example of projection: using the dot product to project one vector onto another. In our current case, we want to project the points that make up the vertices of an object onto a plane, called the *projection plane* or the *view plane*. We do this by following a *line of projection* through each point and determining where it hits the plane. These lines could be perpendicular to the plane, but as we'll see, they don't have to be.

To understand how this works, we'll look at a very old form of optical projection known as the *camera obscura* (Latin for "dark room"). Suppose one enters a darkened room on a sunny day, and there is a small hole allowing a fraction of sunlight to enter the room. This light will be projected onto the opposite wall of the room, displaying an image of the world outside, albeit upside down and flipped left to right (Figure 6.5). This is the same principle that allows a pinhole camera to work; the hole is acting like the focal point of a lens. In this case, all the lines of projection pass through a single *center of projection*. We can determine where a point will project to on the plane by constructing a line through both the original point and the center of projection and calculating where it will intersect the plane of projection. The virtual film in this case is a rectangle on the view plane, known as the *view window*. This will eventually get mapped to our display.

This sort of projection is known as *perspective projection*. Note that this relates to our perceived view in the real world. As an object moves farther away, its corresponding projection will shrink on the projection plane. Similarly, lines that are parallel in view space will appear to converge as their extreme points move farther away from the view position. This gives us a result consistent with our expected view in the real world. If we stand on some railroad tracks and look down a straight section, the rails will converge in the distance, and the ties will appear to shrink in size and become closer together. In most cases, since we are rendering real-world scenes — or at least, scenes that we want to be perceived as real world — this will be the projection we will use.

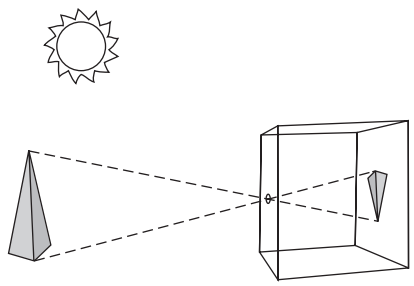


Figure 6.5 Camera obscura.

There is, of course, one minor problem: The projected image is upside down and backwards. One possibility is just to flip the image when we display it on our medium. This is what happens with a camera: The image is captured on film upside down, but we can just rotate the negative or print to view it properly. This is not usually done in graphics. Instead, the projection plane is moved to the other side of the center of projection, which is now treated as our view position (Figure 6.6). As we'll see, the mathematics for projection in this case are quite simple, and the objects located in the forward direction of our view will end up being projected right-side up. The objects behind the view will end up projecting upside down, but (a) we don't want to render them anyway, and (b) as we'll see, there are ways of handling this situation.

An alternate type of projection is *parallel projection*, which can be thought of as a perspective projection where the center of projection is infinitely distant. In this case, the lines of projection do not converge; they always remain parallel (Figure 6.7), hence the name. The placement of the view position and view plane is irrelevant in this case, but we place them in the same relative location to maintain continuity with perspective projection.

Parallel projection produces a very odd view if used for a scene: Objects remain the same size no matter how distant they are, and parallel lines remain parallel. Parallel projections are usually used for computer-assisted design (CAD) programs, where maintaining parallel lines is important. They are also useful for rendering 2D elements like interfaces; no matter how far from the eye a model is placed, it always will be the same size, presumably the size we expect.

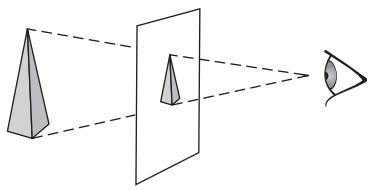


Figure 6.6 Perspective projection.

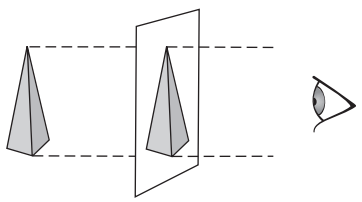


Figure 6.7 Orthographic parallel projection.

A parallel projection where the lines of projection are perpendicular to the view plane is called an *orthographic projection*. By contrast, if they are not perpendicular to the view plane, this is known as an *oblique projection* (Figure 6.8). Two common oblique projections are the *cavalier projection*, where the projection angle is 45 degrees, and the *cabinet projection*, where the projection angle is $\cot^{-1}(1/2)$. When using cavalier projections, projected lines have the same length as the original lines, so there is no perceived foreshortening. This is useful when printing blueprints, for example, as any line can be measured to find the exact length of material needed to build the object. With cabinet projections, lines perpendicular to the projection plane foreshorten to half their length (hence the $\cot^{-1}(1/2)$), which gives a more realistic look without sacrificing the need for parallel lines.

We can also have oblique perspective projections where the line from the center of the view window to the center of projection is not perpendicular to the view plane. For example, suppose we need to render a mirror. To do so, we'll render the space using a plane reflection transformation and clip it to the boundary of the mirror. The plane of the mirror is our projection plane, but it may be at an angle to our view direction (Figure 6.9). For now, we'll concentrate on constructing projective transformations perpendicular to the projection plane and examine these special cases later.

As a side note, oblique projections can occur in the real world. The classic pictures we see of tall buildings, shot from the ground but with parallel sides,

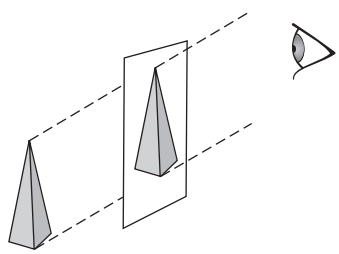


Figure 6.8 Oblique parallel projection.

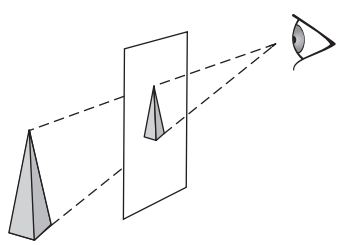


Figure 6.9 Oblique perspective projection.

are done with a "view camera." This device has an accordion-pleated hood that allows the photographer to bend and tilt the lens up while keeping the film parallel to the side of the building. Ansel Adams also used such a camera to capture some of his famous landscape photographs.

6.3.2 Normalized Device Coordinates

Before we begin projecting, our objects have passed through the view stage of the pipeline and so are in view frame coordinates. We will be projecting from this space in $\mathbb{R}^3$ to the view plane, which is in $\mathbb{R}^2$. In order to accomplish this, it will be helpful to define a frame for the space of the view plane. We'll use as our origin the center of the view window, and create basis vectors that align with the sides of the view window, with magnitudes of half the width and height of the window, respectively (Figure 6.10(a)). Within this frame, our view window is transformed into a square two units wide and centered at the origin, bounded by the $x = 1$, $x = -1$, $y = 1$, and $y = -1$ lines (Figure 6.10(b)).

Using this as our frame provides a certain amount of flexibility when mapping to devices of varying size. Rather than transform directly to our screen area, which could be of variable width and height, we use this normalized form as an intermediate step to simplify our calculations and then do the screen conversion as our final step. Because of this, coordinates in this frame are known as *normalized device coordinates*.

To take advantage of the normalized device coordinate frame, or *NDC space*, we'll want to create our projection so that it always gives us the -1 to 1 behavior, regardless of the exact view configuration. This helps us to compartmentalize the process of projection (just as the view matrix did for viewing). When we're done projecting, we'll stretch and translate our NDC values to match the width and height of our display.

To simplify this mapping to the NDC frame, we will begin by using a view window in the view frame with a height of two units. This means that for the case of a centered view window, xy coordinates on the view plane will be equal to the projected coordinates in the NDC frame. In this way we can consider the projection as related to the view plane in view coordinates and not worry about a subsequent transformation.

6.3.3 View Frustum

The question remains: How do we determine what will lie within our view window? We could, naively, project all of the objects in the world to the view plane and then, when converting them to pixels, ignore those pixels that lie outside of the view window. However, for a large number of objects this would be very inefficient. It would be better to constrain our space to a convex volume,

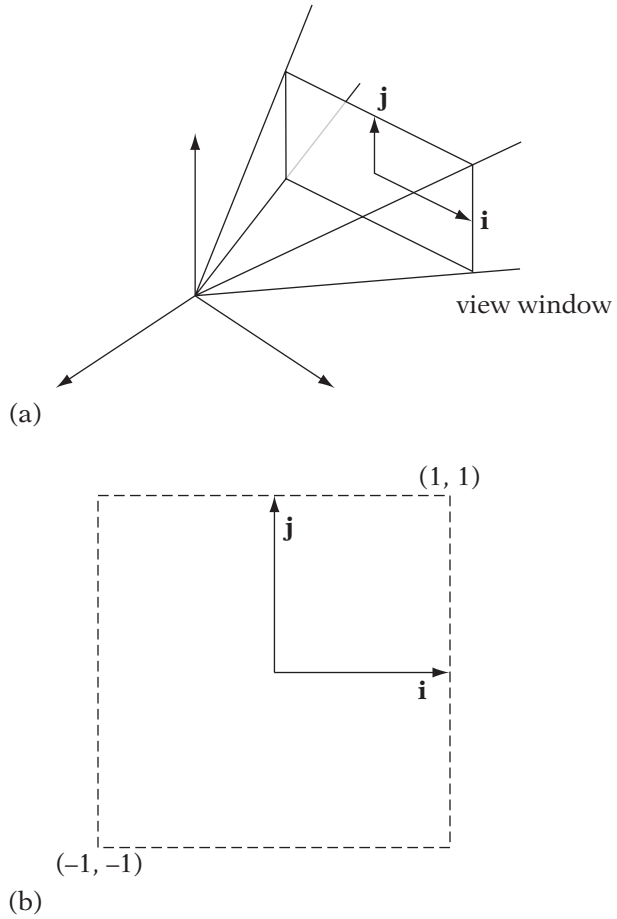


FIGURE 6.10 (a) NDC frame in view window, and (b) view window after NDC transformation.

specified by a set of six planes. Anything inside these planes will be rendered; everything outside them will be ignored. This volume is known as the *view frustum*, or *view volume*.

To constrain what we render in the view frame xy directions, we specify four planes aligned with the edges of the view window. For perspective projection each plane is specified by the view position and two adjacent vertices of the view window (Figure 6.11), producing a semi-infinite pyramid. The angle between the upper plane and the lower plane is called the vertical *field of view*.

There is a relationship between field of view, view window size, and view plane distance: Given two, we can easily find the third. For example, we can fix the view window size, adjust the field of view, and then compute the distance

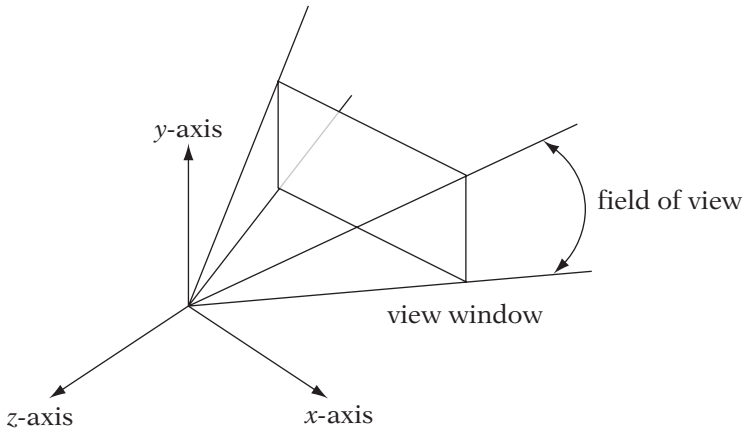


Figure 6.11 Perspective view frustum (right-handed system).

to the view plane. As the field of view gets larger, the distance to the view plane needs to get smaller to maintain the view window size. Similarly, a small field of view will lead to a longer view plane distance. Alternatively, we can set the distance to the view plane to a fixed value and use the field of view to determine the size of our view window. The larger the field of view, the larger the window and the more objects are visible in our scene. This gives us a primitive method for creating telephoto (narrow field of view) or wide-angle (wide field of view) lenses. We will discuss the relationship among these three quantities in more detail when we cover perspective projection.

In our case, the view window size is fixed, so when adjusting our field of view, we will move the view plane relative to the center of projection. This continues to match our camera analogy: The film size is fixed and the lens moves in and out to create a telephoto or wide-angle effect.

Usually the field of view chosen needs to match the display medium, as the user perceives it, as much as possible. For a standard monitor placed about three feet away, the monitor only covers about a 25- to 30-degree field of view from the perspective of the user, so we would expect that we would use a field of view of that size in the game. However, this constrains the amount we can see in the game to a narrow area, which feels unnatural because we're used to a 180-degree field of view in the real world. The usual compromise is to set the field of view to the range of 60–90 degrees. The distortion is not that perceptible and it allows the user to see more of the game world. If the monitor were stretched to cover more of your personal field of view, as in a widescreen monitor or some virtual reality systems, a larger field of view would be appropriate. And of course, if the desired effect is of a telephoto or wide-angle lens, a narrower or wider field of view, respectively, is appropriate.

For parallel projection, the *xy* culling planes are parallel to the direction of projection, so opposite planes are parallel and we end up with a parallelopiped that is open at two ends (Figure 6.12). There is no concept of field of view in this case.

In both cases, to complete a closed view frustum we also define two planes that constrain objects in the view frame *z*-direction: the near and far planes (Figure 6.13). With perspective projection it may not be obvious why we need a near plane, since the *xy* planes converge at the center of projection, closing the viewing region at that end. However, as we will see when we start talking about the perspective transformation, rendering objects at the view frame origin (which in our case is the same as the center of projection) can lead to a possible division by zero. This would adversely affect our rendering process. We could also, like some viewing systems, use the view plane as the near plane, but not doing so allows us a little more flexibility.

In some sense, the far plane is optional. Since we don't have an infinite number of objects or an infinite amount of game space, we could forego using the far plane and just render everything within the five other planes. However, the far plane is useful for culling objects and area from our rendering process, so having a far plane is good for efficiency's sake. It is also extremely important in the hidden surface removal method of *z*-buffering; the distance between the near and far planes is a factor in determining the precision we can expect in our *z* values. We'll discuss this in more detail in Chapter 9.

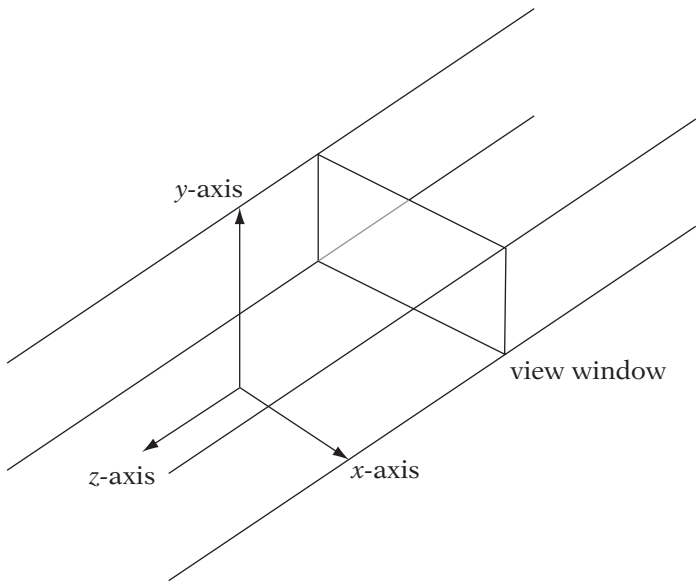


FIGURE 6.12 Parallel view frustum (right-handed system).

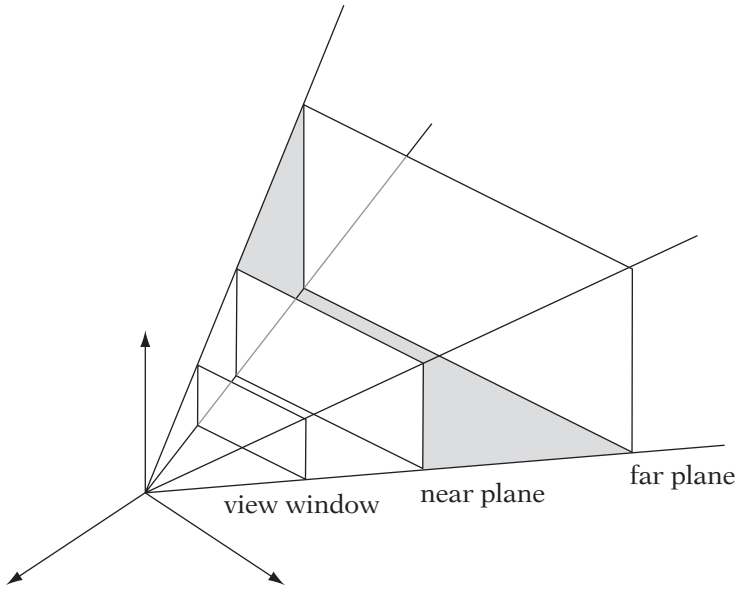


Figure 6.13 View frustum with near plane and far plane.

6.3.4 Homogeneous Coordinates

There is one more topic we need to cover before we can start discussing projection. Previously we stated that a point in $\mathbb{R}^3$ can be represented by $(x, y, z, 1)$ without explaining much about what that might mean. This representation is part of a more general representation for points known as homogeneous coordinates, which prove useful to us when handling perspective projections. In general, homogeneous coordinates work as follows: If we have a "standard" representation in n-dimensional space, then we can represent the same point in a $(n + 1)$–dimensional space by scaling the original coordinates by a single value and then adding the scalar to the end as our final coordinate. Since we can choose from an infinite number of scalars, a single point in $\mathbb{R}^n$ will be represented by an infinite number of points in the $(n + 1)$–dimensional space. This $(n + 1)$–dimensional space is called a *real projective space* or $\mathbb{R}P^n$. In computer graphics parlance, the real projective space $\mathbb{R}P^3$ is also often called *homogeneous space*.

Suppose we start with a point (x, y, z) in $\mathbb{R}^3$, and we want to map it to a point (x', y', z', w) in homogeneous space. We pick a scalar for our fourth element w, and scale the other elements by it, to get (xw, yw, zw, w). As we might expect, our standard value for w will be 1, so (x, y, z) maps to $(x, y, z, 1)$. To map back to 3D space, divide the first three coordinates by w, so (x', y', z', w) goes to $(x'/w, y'/w, z'/w)$. Since our standard value for w is just 1, we could

just drop the w: $(x', y', z', 1) \rightarrow (x', y', z')$. However, in the cases that we'll be concerned with next, we need to perform the division by w.

What happens when $w = 0$? In this case, a point in $\mathbb{R}P^3$ doesn't represent a point in $\mathbb{R}^3$, but a vector. We can think of this as a "point at infinity." While we will try to avoid cases where $w = 0$, they do creep in, so checking for this before performing the homogeneous division is often wise.

6.3.5 Perspective Projection

SOURCE CODE DEMO Perspective

Since this is the most common projective transform we'll encounter, we'll begin by constructing the mathematics necessary for the perspective projection. To simplify things, let's take a 2D view of the situation on the yz plane and ignore the near and far planes for now (Figure 6.14). We have the y-axis pointing up, as in the view frame, and the projection direction along the negative z-axis as it would be in OpenGL. The point on the left represents our center of projection, and the vertical line our view plane. The diagonal lines represent our y culling planes.

Suppose we have a point P_v in view coordinates that lies on one of the view frustum planes, and we want to find the corresponding point P_s that lies on the view plane. Finding the y coordinate of P_s is simple: We follow the line of projection along the plane until we hit the top of the view window. Since the height of the view window is 2 and is centered on 0, the y coordinate of P_s is half the height of the view window, or 1. The z coordinate will be negative since we're looking along the negative z-axis and will have a magnitude equal to the distance d from the view position to the projection plane. So, the z coordinate will be $-d$.

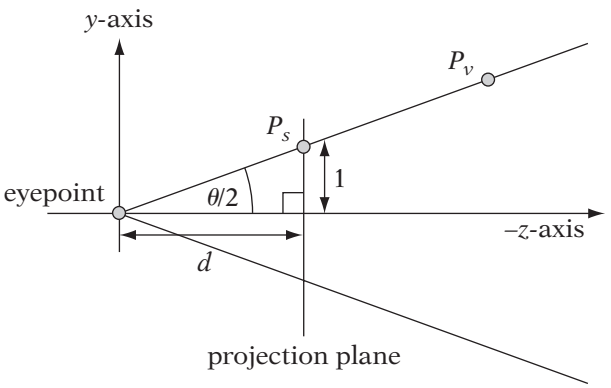


Figure 6.14 Perspective projection construction.

But how do we compute d? As we see, the cross section of the y view frustum planes are represented as lines from the center of projection through the extents of the view window $(1, d)$ and $(-1, d)$. The angle between these lines is our field of view θ_{fov}. We'll simplify things by considering only the area that lies above the negative z-axis; this bisects our field of view to an angle of $\theta_{fov}/2$. If we look at the triangle bounded by the negative z-axis, the cross section of the upper view frustum plane, and the cross section of the projection plane, we can use trigonometry to compute d. Since we know the distance between the negative z-axis and the extreme point P_s is 1, we can say that

$$\frac{1}{d} = \tan(\theta_{fov}/2)$$

Rewriting this in terms of d, we get

$$\begin{aligned} d &= \frac{1}{\tan\left(\frac{\theta_{fov}}{2}\right)} \\ &= \cot\left(\frac{\theta_{fov}}{2}\right) \end{aligned}$$

So for this fixed-view window size, as long as we know the angle of field of view, we can compute the distance d, and vice versa.

This gives the coordinates for any point that lies on the upper y view frustum plane; in this 2D cross section they all project down to a single point $(1, -d)$. Similarly, points that lie on the lower y frustum plane will project to $(-1, -d)$. But suppose we have a general point (y_v, z_v) in view space. We know that its projection will lie on the view plane as well, so its z_{ndc} coordinate will be $-d$. But how do we find y_{ndc}?

We can compute this by using similar triangles (Figure 6.15). If we have a point (y_v, z_v), the length of the sides of the corresponding right triangle in our diagram are y_v and $-z_v$ (since we're looking down the $-z$-axis, any visible z_v is negative, so we need to negate it to get a positive value). The length of sides of the right triangle for the projected point are y_{ndc} and d. By similar triangles (both have the same angles), we get

$$\frac{y_{ndc}}{d} = \frac{y_v}{-z_v}$$

Solving for y_{ndc}, we get

$$y_{ndc} = \frac{dy_v}{-z_v}$$

This gives us the coordinate in the y direction. If our view region was square, then we could use the same formula for the x direction. Most, however,

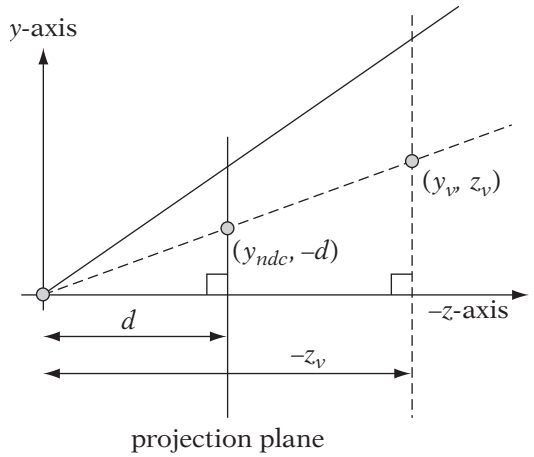


FIGURE 6.15 Perspective projection similar triangles.

are rectangular to match the relative dimensions of a computer monitor or other viewing device. We must correct for this by the aspect ratio of the view region. The aspect ratio a is defined as

$$a = \frac{w_v}{h_v}$$

where w_v and h_v are the width and height of the view rectangle, respectively. We're going to assume that the NDC view window height remains at 2 and correct the NDC view width by the aspect ratio. This gives us a formula for similar triangles of

$$\frac{ax_{ndc}}{d} = \frac{x_v}{-z_v}$$

Solving for x_{ndc}:

$$x_{ndc} = \frac{dx_v}{-az_v}$$

So, our final projection transformation equations are

$$x_{ndc} = \frac{dx_v}{-az_v}$$
$$y_{ndc} = \frac{dy_v}{-z_v}$$

The first thing to notice is that we are dividing by a z coordinate, so we will not be able to represent the entire transformation by a matrix operation, since it is neither linear nor affine. However, it does have some affine elements — scaling by d and d/a, for example — which can be performed by a transformation matrix. This is where the conversion from homogeneous space comes in. Recall that to transform from $\mathbb{R}P^3$ to $\mathbb{R}^3$ we need to divide the other coordinates by the w value. If we can set up our matrix to map $-z_v$ to our w value, we can take advantage of the homogeneous divide to handle the nonlinear part of our transformation. We can write the situation before the homogeneous divide as a series of linear equations:

$$\begin{aligned} x' &= \frac{d}{a}x \\ y' &= dy \\ z' &= dz \\ w' &= -z \end{aligned}$$

and treat this as a four-dimensional (4D) linear transformation. Looking at our basis vectors, $\mathbf{e}_0$ will map to $(d/a, 0, 0, 0)$, $\mathbf{e}_1$ to $(0, d, 0, 0)$, $\mathbf{e}_2$ to $(0, 0, d, -1)$, and $\mathbf{e}_3$ to $(0, 0, 0, 0)$, since w is not used in any of the equations.

Based on this, our homogeneous perspective matrix is

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

As expected, our transformed w value no longer will be 1. Also note that the right-most column of this matrix is all zeros, which means that this matrix has no inverse. This is to be expected, since we are losing one dimension of information. Individual points in view space that lie along the same line of projection will project to a single point in NDC space. Given only the points in NDC space, it would be impossible to reconstruct their original positions in view space.

Let's see how this matrix works in practice. If we multiply it by a generic point in view space, we get

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} dx_v/a \\ dy_v \\ dz_v \\ -z_v \end{bmatrix}$$

Dividing out the w (also called the reciprocal divide), we get

$$x_{ndc} = \frac{dx_v}{-az_v}$$
$$y_{ndc} = \frac{dy_v}{-z_v}$$
$$z_{ndc} = -d$$

which is what we expect.

So far, we have dealt with projecting x and y and completely ignored z. In the preceding derivation all z values map to $-d$, the negative of the distance to the projection plane. While losing a dimension makes sense conceptually (we are projecting from a 3D space down to a 2D plane, after all), for practical reasons it is better to keep some measure of our z values around for z-buffering and other depth comparisons (discussed in more detail in Chapter 9). Just as we're mapping our x and y values within the view window to an interval of $[-1, 1]$, we'll do the same for our z values within the near plane and far plane positions. We'll specify the near and far values n and f relative to the view position, so points lying on the near plane have a z_v value of $-n$, which maps to a z_{ndc} value of -1. Those points lying on the far plane have a z_v value of $-f$ and will map to 1 (Figure 6.16).

We'll derive our equation for z_{ndc} in a slightly different way than our xy coordinates. There are two parts to mapping the interval $[-n, -f]$ to $[-1, 1]$. The first is scaling the interval to a width of 2, and the second is translating it to $[-1, 1]$. Ordinarily, this would be a straightforward linear process, however, we also have to contend with the final w divide. Instead, we'll create

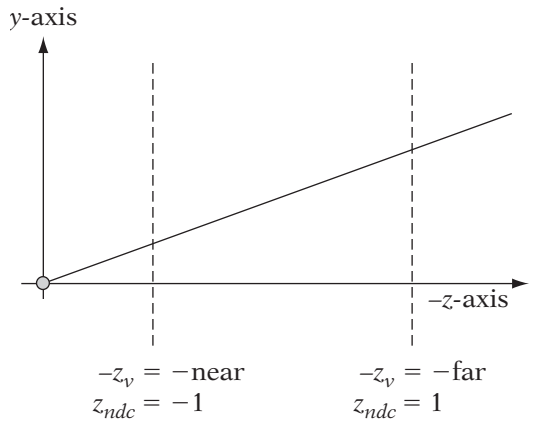


FIGURE 6.16 Perspective projection: z values.

a perspective matrix with unknowns for the scaling and translation factors and use the fact that we know the final values for $-n$ and $-f$ to solve for the unknowns. Our starting perspective matrix, then, is

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where A and B are our unknown scale and translation factors, respectively. If we multiply this by a point $(0, 0, -n)$ on our near plane, we get

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -n \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -An + B \\ n \end{bmatrix}$$

Dividing out the w gives

$$z_{ndc} = -A + \frac{B}{n}$$

We know that any point on the near plane maps to a normalized device coordinate of -1, so we can substitute -1 for z_{ndc} and solve for B, which gives us

$$B = (A - 1)n \tag{6.3}$$

Now we'll substitute equation 6.3 into our original matrix and multiply by a point $(0, 0, -f)$ on the far plane:

$$\begin{bmatrix} d/a & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & A & (A-1)n \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -f \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -Af + (A-1)n \\ f \end{bmatrix}$$

This gives us a z_{ndc} of

$$\begin{aligned} z_{ndc} &= -A + (A-1)\frac{n}{f} \\ &= -A + A\left(\frac{n}{f}\right) - \frac{n}{f} \\ &= A\left(\frac{n}{f} - 1\right) - \frac{n}{f} \end{aligned}$$

Setting z_{ndc} to 1 and solving for A, we get

$$\begin{aligned} A\left(\frac{n}{f}-1\right)-\frac{n}{f} &= 1 \\ A\left(\frac{n}{f}-1\right) &= \frac{n}{f}+1 \\ A &= \frac{\frac{n}{f}+1}{\frac{n}{f}-1} \\ &= \frac{n+f}{n-f} \end{aligned}$$

If we substitute this into equation 6.3, we get

$$B = \frac{2nf}{n-f}$$

So, our final perspective matrix is

$$\mathbf{M}_{persp} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The matrix that we have generated is the same one produced by an OpenGL call: `gluPerspective()`. This function takes the field of view,[1] aspect ratio, and near and far plane settings, builds the perspective matrix, and multiplies it by the current matrix.

It is important to be aware that this matrix will not work for all viewing systems. For one thing, for most other viewing systems (i.e., other than OpenGL), our view frame looks down the positive z-axis, so this affects both our xy and z transformations. For example, in this case we have mapped $[-n, -f]$ to $[-1, 1]$. With the standard system we would want to begin by mapping $[n, f]$ to the NDC z range. In addition, this range is not always set to $[-1, 1]$. Direct3D, for one, has a default mapping of to $[0, 1]$ in the z direction.

1. Recall that our value d is generated from the field of view by $d = \cot(\theta_{fov}/2)$.

Using the standard view frame and this mapping gives us a perspective transformation matrix of

$$\mathbf{M}_{pD3D} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix can be derived using the same principles described above.

When setting up a perspective matrix, it is good to be aware of the issues involved in rasterizing z values. In particular, to maintain z precision keep the near and far planes as close together as possible. More details on managing perspective z precision can be found in Chapter 9.

6.3.6 Oblique Perspective

The matrix we constructed in the previous section is an example of a standard perspective matrix, where the direction of projection through the center of the view window is perpendicular to the view plane. A more general example of perspective is generated by the OpenGL `glFrustum()` call. This call takes six parameters: the near and far z distances, as before, and four values that define our view window on the near z plane: the x interval $[l, r]$ (left, right) and the y interval $[b, t]$ (bottom, top). Figure 6.17(a) shows how this looks in $\mathbb{R}^3$, and Figure 6.17(b) shows the cross section on the yz plane. As we can see, these values need not be centered around the z-axis, so we can use them to generate an oblique projection.

To derive this matrix, once again we begin by considering similar triangles in the y direction. Remember that given a point $(y_v, -z_v)$, we project to a point on the view plane $(dy_v/-z_v, -d)$, where d is the distance to the projection. However, since we're using our near plane as our projection plane, this is just $(ny_v/-z_v, -n)$. The projection remains the same, we're just moving the window of projected points that lie within our view frustum.

With our previous derivation, we could stop at this point because our view window on the projection plane was already in the interval $[-1, 1]$. However, our new view window lies in the interval $[b, t]$. We'll have to adjust our values to properly end up in NDC space. The first step is to translate the center of the window, located at $(t + b)/2$, to the origin. Applying this translation to the current projected y coordinate gives us

$$y' = y - \frac{(t + b)}{2}$$

We now need to scale to change our interval from a magnitude of $(t - b)$ to a magnitude of 2 by using a scale factor $2/(t - b)$:

$$y_{ndc} = \frac{2y}{t-b} - \frac{2(t+b)}{2(t-b)} \tag{6.4}$$

If we substitute $ny_v/-z_v$ for y and simplify, we get

$$y_{ndc} = \frac{2n\dfrac{y_v}{-z_v}}{t-b} - \frac{2(t+b)}{2(t-b)}$$

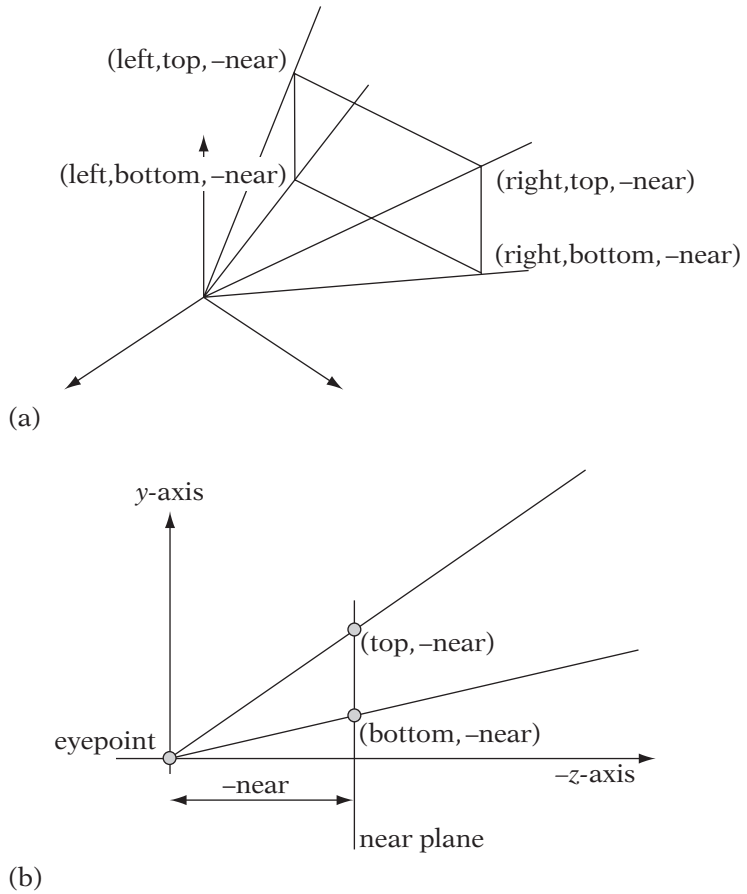


FIGURE 6.17 (a) View window for `glFrustum`, 3D view. (b) View window for `glFrustum`, cross section.

$$
\begin{aligned}
&= \frac{2n\dfrac{y_v}{-z_v}}{t-b} - \frac{(t+b)\dfrac{-z_v}{-z_v}}{t-b} \\
&= \frac{1}{-z_v}\left(\frac{2n}{t-b}y_v + \frac{t+b}{t-b}z_v\right)
\end{aligned}
$$

A similar process gives us the following for the x direction:

$$
x_{ndc} = \frac{1}{-z_v}\left(\frac{2n}{r-l}x_v + \frac{r+l}{r-l}z_v\right)
$$

We can use the same A and B from our original perspective matrix, so our final projection matrix is

$$
\mathbf{M}_{oblpersp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}
$$

A casual inspection of this matrix gives some sense of what's going on here. We have a scale in the x, y, and z directions, which provides the mapping to the interval $[-1, 1]$. In addition, we have a translation in the z direction to align our interval properly. However, in the x and y directions, we are performing a z-shear to align the interval, which provides us with the oblique projection.

The equivalent Direct3D matrix is

$$
\mathbf{M}_{opD3D} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}
$$

As unusual as it might appear, there are a number of applications of oblique perspective projection in real-time graphics. First of all, it can be used in mirrors: We treat the mirror as our view window, the mirror plane as our view plane, and the viewer's location as our view position. If we apply a plane reflection to all of our objects, flipping them around the mirror plane, and then render with the appropriate visual effects, we will end up with a result in the view window that emulates a mirror.

Another application is stereo. By using a single view plane and view window, but separate view positions for each eye that are offset from the standard center of projection, we get slightly different projections of the world. By using either a red-blue system to color each view differently, or some sort of

goggle system that displays the left and right views in each eye appropriately, we can provide a good approximation of stereo vision. We have included an example of this on the CD-ROM.

Finally, this can be used for a system called *fishtank VR*. Normally we think of VR as a helmet attached to someone's head with a display for each eye. However, by attaching a tracking device to a viewer's head we can use a single display and create an illusion that we are looking through a window into a world on the other side. This is much the same principle as the mirror: The display is our view window and the tracked location of the eye is our view position. Add stereo and this gives a very pleasing effect.

6.3.7 Orthographic Parallel Projection

Source Code Demo Orthographic

After considering perspective projection in two forms, orthographic projection is much easier. Examine Figure 6.18, which shows a side view of our projection space as before, with the lines of projection passing through the view plane and the near and far planes shown as vertical lines. This time the lines of projection are parallel to each other (hence this is a parallel projection) and parallel to the z-axis (hence an orthographic projection).

We can use this to help us generate the matrix for the OpenGL `glOrtho()` call. Like `glFrustum()`, this call takes six parameters: the near and far z distances, and four values l, r, b, and t that define our view window on the near z plane. As before, the near plane is our projection plane, so a point (y_v, z_v) projects to a point $(y_v, -n)$. Note that since this is a parallel projection, there is no division by z or scale by d; we just use the y value directly. Like `glFrustum()` we now need to consider only values between t and b and scale and translate

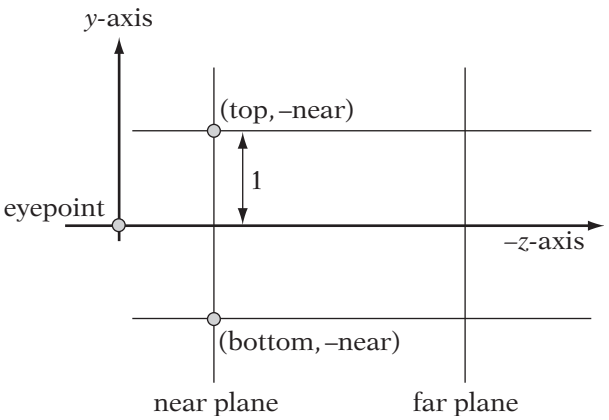


Figure 6.18 Orthographic projection construction.

them to the interval $[-1, 1]$. Substituting y_v into our range transformation equation 6.4, we get

$$y_{ndc} = \frac{2y_v}{t-b} - \frac{t+b}{t-b}$$

A similar process gives us the equation for x_{ndc}. We can do the same for z_{ndc}, but since our viewable z values are negative and our values for n and f are positive, we need to negate our z value and then perform the range transformation. The result of all three equations is

$$\mathbf{M}_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

There are a few things we can notice about this matrix. First of all, multiplying by this matrix gives us a w value of 1, so we don't need to perform the homogeneous division. This means that our z values will remain linear; that is, they will not compress as they approach the far plane. This gives us better z resolution at far distances than the perspective matrices. It also means that this is a linear transformation matrix and possibly invertible.

Secondly, in the x and y directions, what was previously a z-shear in the oblique perspective matrix has become a translation. Before, we had to use shear, because for a given point the displacement was dependent on the distance from the view position. Because the lines of projection are now parallel, all points displace equally, so only a translation is necessary.

The Direct3D equivalent matrix is

$$\mathbf{M}_{orthoD3D} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.3.8 Oblique Parallel Projection

While most of the time we'll want to use orthographic projection, we may from time to time need an oblique parallel projection. For example, suppose for part of our interface we wish to render our world as a set of schematics or display particular objects with a 2D CAD/CAM feel. This set of projections will achieve our goal.

Neither OpenGL nor Direct3D has a particular routine that handles oblique parallel projections, so we'll derive one ourselves. We will give our projection a slight oblique angle ($\cot^{-1}(1/2)$, which is about 63.4 degrees), which gives a 3D look without perspective. More extreme angles in x and y tend to look strangely flat.

Figure 6.19 is another example of our familiar cross section, this time showing the lines of projection for our oblique projection. As we can see, we move one unit in the y direction for every two units we move in the z direction. Using the formula of $\tan(\theta) = opposite/adjacent$, we get

$$\tan(\theta) = \frac{2}{1}$$
$$\cot(\theta) = \frac{1}{2}$$
$$\theta = \cot^{-1}\frac{1}{2}$$

which confirms the expected value for our oblique angle.

As before, we'll consider the yz case first and extrapolate to x. Moving one unit in y and two units in $-z$ gives us the vector $(1, -2)$, so the formula for the line of projection for a given point P is

$$L(t) = P + t(1, -2)$$

We're only interested in where this line crosses the near plane, or where

$$P_z - 2t = -n$$

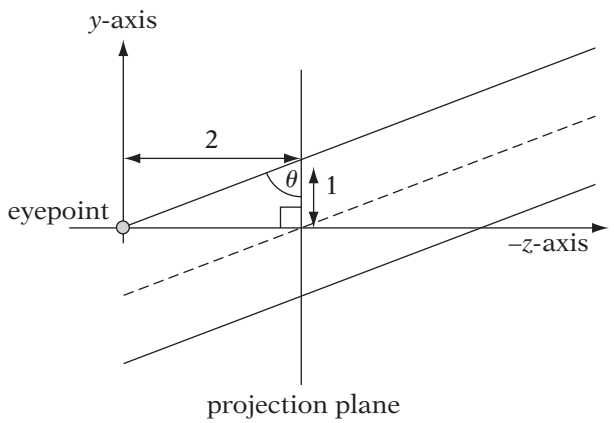


FIGURE 6.19 Example of oblique parallel projection.

Solving for t, we get

$$t = \frac{1}{2}(n + P_z)$$

Plugging this into the formula for the y coordinate of $L(t)$, we get

$$y' = P_y + \frac{1}{2}(n + P_z)$$

Finally, we can plug this into our range transformation equation 6.4 as before to get

$$y_{ndc} = 2\frac{\left[y_v + \frac{1}{2}(n + z_v)\right]}{t - b} - \frac{t + b}{t - b}$$

$$= \frac{2y_v}{t - b} - \frac{t + b}{t - b} + \frac{z_v + n}{t - b}$$

Once again, we examine our transformation equation more carefully. This is the same as the orthographic transformation we had before, with an additional z-shear, as we'd expect for an oblique projection. In this case, the shear plane is the near plane rather than the xy plane, so we add an additional factor of $\frac{n}{t-b}$ to take this into account.

A similar process can be used for x. Since the oblique projection has a z-shear, z is not affected and so,

$$\mathbf{M}_{obl} = \begin{bmatrix} \frac{2}{r-l} & 0 & \frac{1}{r-l} & -\frac{r+l-n}{r-l} \\ 0 & \frac{2}{t-b} & \frac{1}{t-b} & -\frac{t+b-n}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{n+f}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Direct3D equivalent matrix is

$$\mathbf{M}_{oblD3D} = \begin{bmatrix} \frac{2}{r-l} & 0 & -\frac{1}{r-l} & -\frac{r+l-n}{r-l} \\ 0 & \frac{2}{t-b} & -\frac{1}{t-b} & -\frac{t+b-n}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.4 Culling and Clipping

6.4.1 Why Cull or Clip?

We will now take a detour from discussing the transformation aspect of our pipeline to discuss a process that often happens at this point in many renderers. In order to improve rendering, both for speed and appearance's sake, it is necessary to cull and clip objects. Culling is the process of removing objects from consideration for some process, whether it be rendering, simulation, or collision detection. In this case, that means we want to ignore any models or whole pieces of geometry that lie outside of the view frustum, since they will never end up being projected to the view window. In Figure 6.20, the lighter objects lie outside of the view frustum and so will be culled for rendering.

Clipping is the process of cutting geometry to match a boundary, whether it be a polygon or, in our case, a plane. Vertices that lie outside the boundary will be removed and new ones generated for each edge that crosses the boundary. For example, in Figure 6.21 we see a cube being clipped by a plane, showing the extra vertices created where each edge intersects the plane. We'll use this for any models that cross the view frustum, cutting the geometry so that it fits within the frustum. We can think of this as slicing a piece of geometry off for every frustum plane.

Why should we want to use either of these for rendering? For one thing, it is more efficient to remove any data that will not ultimately end up on the screen. While copying the transformed object to the frame buffer (a process called *rasterization*) is almost always done in hardware and thus is fast, it is not free. Anywhere we can avoid unnecessary work is good.

But even if we had infinite rasterization power, we would still want to cull and clip when performing perspective projection. Figure 6.22 shows one

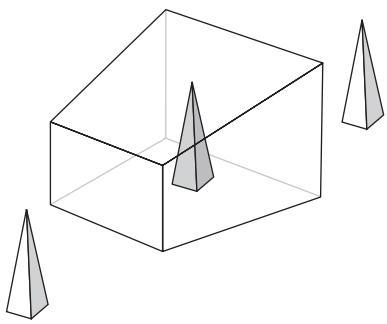


Figure 6.20 View frustum culling.

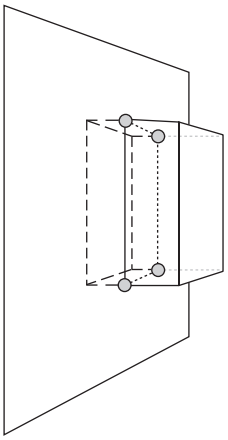


FIGURE 6.21 View frustum clipping.

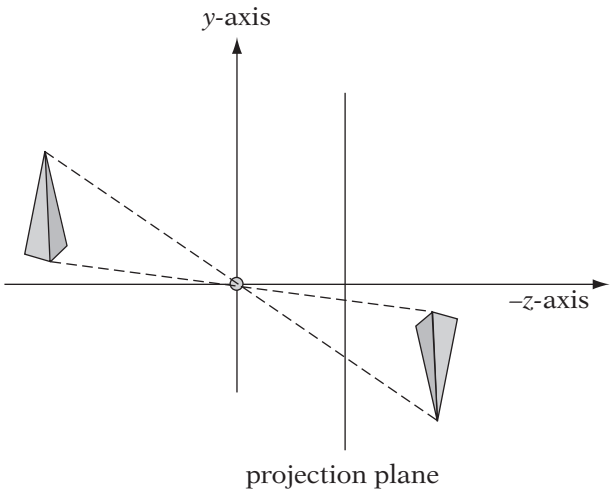


FIGURE 6.22 Projection of objects behind the eye.

example why. Recall that we finessed the problem of the camera obscura inverting images by moving the view plane in front of the center of projection. However, we still have the same problem if an object is behind the view position; it will end up projected upside down. The solution is to cull objects that lie behind the view position.

Figure 6.23(a) shows another example. Suppose we have a polygon edge $\overline{PQ}$ that crosses the $z = 0$ plane. Endpoint P projects to a point P' on the view plane, and Q to Q'. With the correct projection, the intermediate points of the

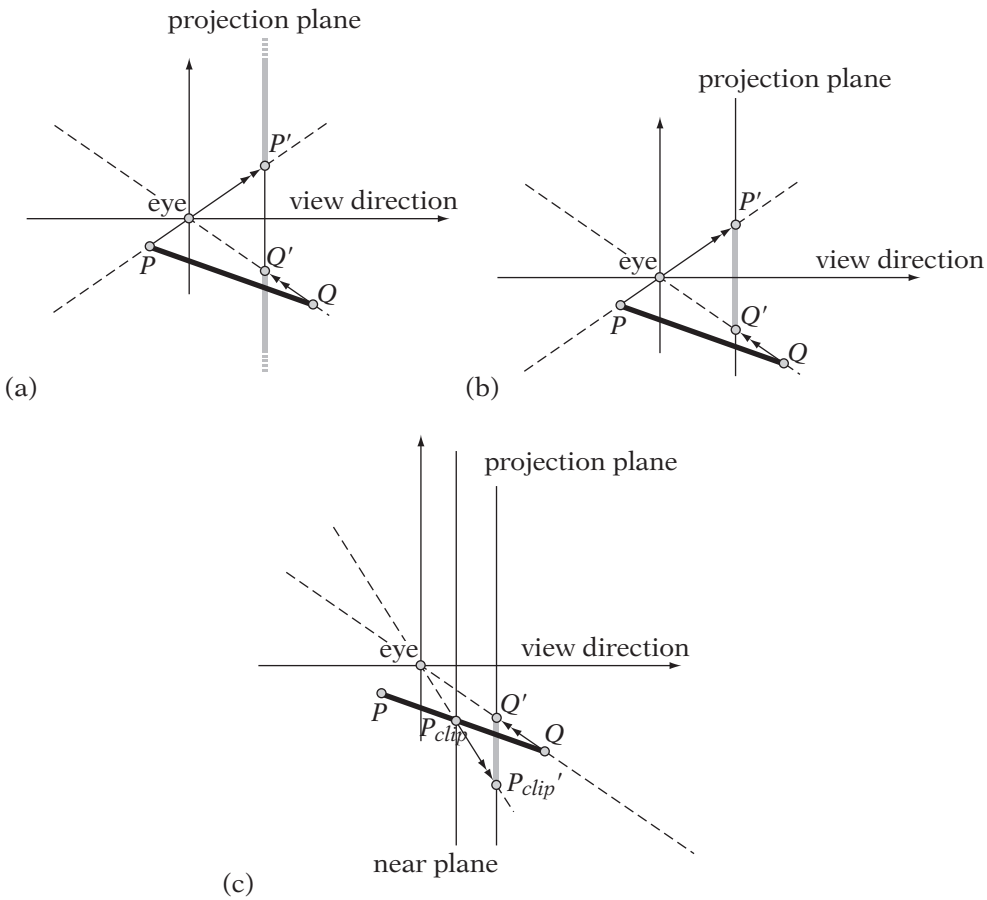


FIGURE 6.23 (a) Projection of line segment crossing behind view point. (b) Incorrect line segment rendering based on projected endpoints. (c) Line segment rendering when clipped to near plane.

line segment should start at the middle of the view, move up, and wrap around to reemerge at the bottom of the view. In practice, however, the rasterizing hardware has only the two projected vertices as input. It will take the vertices and render the shortest line segment between them (Figure 6.23(b)). If we clip the line segment to only the section that is viewable and then project the endpoints (Figure 6.23(c)), we end with only a portion of the line segment, but at least it is from the correct projection.

There is also the problem of vertices that lie on the $z = 0$ plane. When transformed to homogeneous space by the perspective matrix, a point

$(x, y, 0, 1)$ will become $(x', y', z', 0)$. The resulting transformation into NDC space will be a division by 0, which is not valid.

To avoid all of these issues, at the very least we need to set a near plane that lies in front of the eye so that the view position itself does not lie within the view frustum. We first cull any objects that lie on the same side of the near plane as the view position. We then clip any objects that cross the near plane. This avoids both the potential of dividing by 0 (although it is sometimes prudent to check for it anyway, at least in a debug build) and trying to render any line segments passing through infinity.

While clipping to a near plane is a bare minimum, clipping to the top, bottom, left, and right planes is useful as well. While the windowing hardware will usually ignore any pixels that lie outside of a window's visible region (this is commonly known as *scissoring*), it is faster if we can avoid unnecessary rasterization. Also, if we want to set a viewport that covers a subrectangle of a window, not clipping to the border of the viewport may lead to spurious geometry being drawn (although most hardware allows for adjustable scissoring regions; in particular, OpenGL and D3D provide interfaces to set this).

Finally, some hardware has a limited range for screen space positions, for example, 0 to 4095. The viewable area might lie in the center of this range, say from a minimum point of (1728, 1808) to a maximum point of (2688, 2288). The area outside of the viewable area is known as the *guard band*—anything rendered to this will be ignored, since it won't be displayed. In some cases we can avoid clipping in x and y, since we can just render objects whose screen space projection lies within the guard band and know that they will be handled automatically by the hardware. This can improve performance considerably, since clipping can be quite expensive. However, it's not entirely free. Values that lie outside the maximum range for the guard band will wrap around. So, a vertex that would normally project to coordinates that should lie off the screen, say (6096, 6096), will wrap to (2000, 2000)—right in the middle of the viewable area. Unfortunately, the only way to solve this problem is what we were trying to avoid in the first place: clipping in the x and y directions. However, now our clip window encompasses the much larger guard band area, so using the guard band can still reduce the amount of clipping that we have to do overall.

6.4.2 Culling

A naive method of culling a model against the view frustum is to test each of its vertices against each of the frustum planes in turn. We designate the plane normal for each plane as pointing toward the inside half-space. If for one plane $ax + by + cz + d < 0$ for every vertex $P = (x, y, z)$, then the model lies outside of the frustum and we can ignore it. Conversely, if for all the

frustum planes and all the vertices $ax + by + cz + d > 0$, then we know the model lies entirely inside the frustum and we don't need to worry about clipping it.

While this will work, for models with large numbers of vertices this becomes expensive, probably outweighing any savings we might gain by not rendering the objects. Instead, culling is usually done by approximating the object with a convex bounding volume, such as a sphere, that contains all of the vertices for the object. Rather than test each vertex against the planes, we test only the bounding object. Since it is a convex object and all the vertices are contained within it, we know that if the bounding object lies outside of the view frustum, all of the model's vertices must lie outside as well. More information on computing bounding objects and testing them against planes can be found in Chapter 12.

Bounding objects are usually placed in the world frame to aid with collision detection, so culling is often done in the world frame as well. This requires storing a representation of each frustum plane in world coordinates, but the additional 24 values required is worth the speedup gained. We can find each x or y clipping plane in the view frame by using the view position and two corners of the view window to generate the plane. The two z planes (in OpenGL) are $z = -near$ and $z = -far$, respectively. Transforming them to the world frame is a simple case of using the technique for transforming plane normals, as described in Chapter 4.

While view frustum culling can remove a large number of objects from consideration, it's not the only culling method. In Chapter 7 we'll discuss backface culling, which allows us to determine which polygons are pointing away from the camera so we can ignore them. There also are a large number of culling methods that break up the scene in order to cull objects that aren't visible. This can help with interior levels, so you don't render rooms that may be within the view frustum but not visible because they're blocked by a wall. Such methods are out of the purview of this book but are described in detail in many of the references cited in the following sections.

6.4.3 General Plane Clipping

To clip polygons, we first need to know how to clip a polygon edge (i.e., a line segment) to a plane. As we'll see, the problem of clipping a polygon to a plane degenerates to handling this case. Suppose we have a line segment $\overline{PQ}$, with endpoints P and Q, that crosses a plane. We'll say that P is inside our clip space and Q is outside. Our clipped line segment will be $\overline{PR}$, where R is the intersection of the line segment and the plane (Figure 6.24).

To find R, we take the line equation $P + t(Q - P)$, plug it into our plane equation $ax + by + cz + d = 0$, and solve for t. To simplify the equations, we'll define $\mathbf{v} = Q - P$. Substituting the parameterized line coordinates for

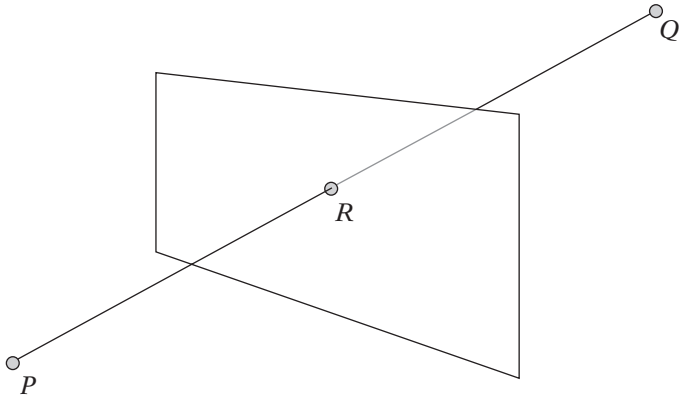


FIGURE 6.24 Clipping edge to plane.

x, y, and z, we get

$$\begin{aligned} 0 &= a(P_x + tv_x) + b(P_y + tv_y) + c(P_z + tv_z) + d \\ &= aP_x + tav_x + bP_y + tbv_y + cP_z + tcv_z + d \\ &= aP_x + bP_y + cP_z + d + t(av_x + bv_y + cv_z) \\ t &= \frac{-aP_x - bP_y - cP_z - d}{av_x + bv_y + cv_z} \end{aligned}$$

And now, substituting in $Q - P$ for $\mathbf{v}$:

$$t = \frac{(aP_x + bP_y + cP_z + d)}{(aP_x + bP_y + cP_z + d) - (aQ_x + bQ_y + cQ_z + d)}$$

We can use Blinn's notation [7], slightly modified, to simplify this to

$$t = \frac{BCP}{BCP - BCQ}$$

where BCP is the result from the plane equation (the boundary coordinate) when we test P against the plane, and BCQ is the result when we test Q against the plane. The resulting clip point R is

$$R = P + \frac{BCP}{BCP - BCQ}(Q - P)$$

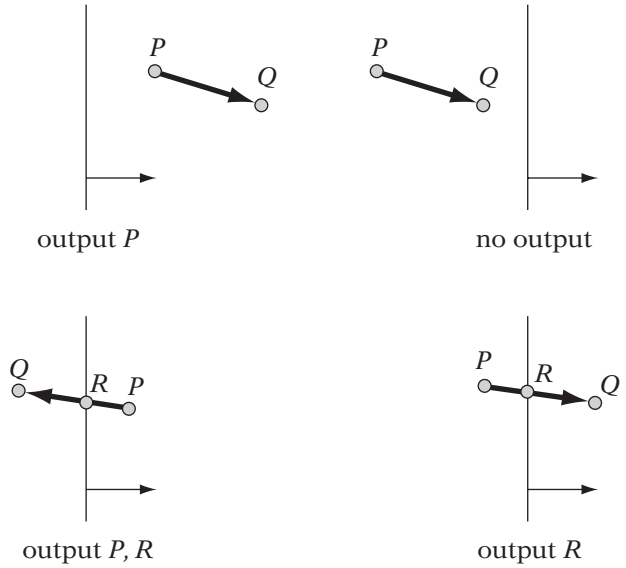


FIGURE 6.25 Four possible cases of clipping an edge against a plane.

To clip a polygon to a plane, we need to clip each edge in turn. A standard method for doing this is to use the Sutherland-Hodgeman algorithm [109]. We first test each edge against the plane. Depending on what the result is, we output particular vertices for the clipped polygon. There are four possible cases for an edge from P to Q (Figure 6.25). If both are inside, then we output P. The vertex Q will be output when we consider it as the start of the next edge. If both are outside, we output nothing. If P is inside and Q is outside, then we compute R, the clip point, and output P and R. If P is outside and Q is inside, then we compute R and output just R—as before, Q will be output as the start of the next edge. The sequence of vertices generated as output will be the vertices of our clipped polygon.

We now have enough information to build a class for clipping vertices, which we'll call `IvClipper`. We can define this as

```
class IvClipper
{
public:
    IvClipper()
    {
        mFirstVertex = true;
    }
    ~IvClipper();
```

```
    void ClipVertex( const IvVector3& end )

    inline void StartClip() { mFirstVertex = true; }
    inline void SetPlane( const IvPlane& plane ) {mPlane = plane;}

private:
    IvPlane   mPlane;   // current clipping plane
    IvVector3 mStart;   // current edge start vertex
    float     mBCStart; // current edge start boundary condition
    bool  mStartInside; // whether current start vertex is inside
    bool  mFirstVertex; // whether expected vertex is start vertex
};
```

Note that `IvClipper::ClipVertex()` takes only one argument: the end vertex of the edge. If we send the vertex pair for each edge down to the clipper, we'll end up duplicating computations. For example, if we clip P_0 and P_1, and then P_1 and P_2, we have to determine whether P_1 is inside or outside twice. Rather than do that, we'll feed each vertex in order to the clipper. By storing the previous vertex (`mStart`) and its plane test information (`mBCStart`) in our `IvClipper` class, we need to calculate data only for the current vertex. Of course, we'll need to prime the pipeline by sending in the first vertex, not treating it as part of an edge, and just storing its boundary information.

Using this, clipping an edge based on the current vertex might look like the following code.

```
void IvClipper::ClipVertex( const IvVector3& end )
{
  float BCend = mPlane.Test(end);
  bool endInside = ( BCend >= 0 );
  if (!mFirstVertex)
  {
     // if one of the points is inside
     if ( mStartInside || endInside )
     {
       // if the start is inside, just output it
       if (mStartInside)
          Output( mStart );
       // if one of them is outside, output clip point
       if ( !(mStartInside && endInside) )
       {
          if (endInside)
          {
            float t = BCend/(BCend - mBCStart);
```

```
            Output( end - t*(end - mStart) );
          }
          else
          {
            float t = mBCStart/(mBCStart - BCend);
            Output( mStart + t*(end - mStart) );
          }
        }
      }
    }

    mStart = end;
    mBCStart = BCend;
    mStartInside = endInside;
    mFirstVertex = false;
  }
```

Note that we generate t in the same direction for both clipping cases — from inside to outside. Polygons will often share edges. If we were to clip the same edge for two neighboring polygons in different directions, we may end up with two slightly different points due to floating-point error. This will lead to visible cracks in our geometry, which is not desirable. Interpolating from inside to outside for both cases avoids this situation.

To clip against the view frustum, or any other convex volume, we need to clip against each frustum plane. The output from clipping against one plane becomes the input for clipping against the next, creating a clipping pipeline. In practice, we don't store the entire clipped polygon, but pass each output vertex down as we generate it. The current output vertex and the previous one are treated as the edge to be clipped by the next plane. The `Output()` call above becomes a `ClipVertex()` for the next stage.

Note that we have only generated new positions at the clip boundary. There are other parameters that we can associate with an edge vertex, such as colors, normals, and texture coordinates (we'll discuss exactly what these are in Chapters 7–9). These will have to be clipped against the boundary as well. We use the same t value when clipping these parameters, so the clip part of our previous algorithm might become as follows.

```
// if one of them is outside, output clip vertex
if ( !(mStartInside && endInside) )
{
    ...
    clipPosition = startPosition + t*(endPosition - startPosition);
    clipColor = startColor + t*(endColor - startColor);
```

```
        clipTexture = startTexture + t*(endTexture - startTexture);
        // Output new clip vertex
    }
```

This is only one example of a clipping algorithm. In most cases, it won't be necessary to write any code to do clipping. The hardware will handle any clipping that needs to be done for rendering. However, for those who have the need or interest, other examples of clipping algorithms are the Liang-Barsky [68], Cohen-Sutherland (found in Foley et al. [38] as well as other graphics texts), and Cyrus-Beck [22] methods. Blinn [8] describes an algorithm for lines that combines many of the features from the previously mentioned techniques; with minor modifications it can be made to work with polygons.

6.4.4 Homogeneous Clipping

In the presentation above, we clip against a general plane. When projecting, however, Blinn and Newell [7] noted that we can simplify our clipping by taking advantage of some properties of our projected points prior to the division by w. Recall that after the division by w, the visible points will have normalized device coordinates lying in the interval $[-1, 1]$, or

$$-1 \leq x/w \leq 1$$
$$-1 \leq y/w \leq 1$$
$$-1 \leq z/w \leq 1$$

Multiplying these equations by w provides the intervals prior to the w division:

$$-w \leq x \leq w$$
$$-w \leq y \leq w$$
$$-w \leq z \leq w$$

In other words, the visible points are bounded by the six planes:

$$w = x$$
$$w = -x$$
$$w = y$$
$$w = -y$$
$$w = z$$
$$w = -z$$

Instead of clipping our points against general planes in the world frame or view frame, we can clip our points against these simplified planes in $\mathbb{R}P^3$ space. For example, the plane test for $w = x$ is $w - x$. The full set of plane tests for a point P are

$$\begin{aligned} BCP_{-x} &= w + x \\ BCP_{x} &= w - x \\ BCP_{-y} &= w + y \\ BCP_{y} &= w - y \\ BCP_{-z} &= w + z \\ BCP_{z} &= w - z \end{aligned}$$

The previous clipping algorithm can be used, with these plane tests replacing the `IvPlane::Test()` call. While these tests are cheaper to compute in software, their great advantage is that since they don't vary with the projection, they can be built directly into hardware, making the clipping process very fast. Because of this, OpenGL clips at two separate stages in the viewing pipeline. After a point is transformed into the view frame, it is clipped against any user-defined clipping planes set by the `glClippingPlane()` call. Then the point is multiplied by the projection matrix, clipped in homogeneous space, and finally the coordinates are divided by w to place the clipped point in the NDC frame.

There is one wrinkle to homogeneous clipping, however. Figure 6.26 shows the visible region for the x coordinate in homogeneous space. However,

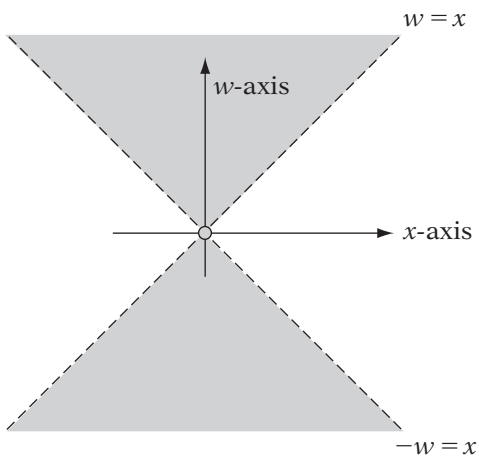


FIGURE 6.26 Homogeneous clip regions for NDC interval [−1,1].

our plane tests will clip to the upper triangle region of that hourglass shape — any points that lie in the lower region will be inadvertently removed. With the projections that we have defined, this will happen only if we use a negative value for the w value of our points. And since we've chosen 1 as the standard w value for points, this shouldn't happen. However, if you do have points that for some reason have negative w values, Blinn [8] recommends the following procedure: transform, clip, and render your points normally; then multiply your projection matrix by -1; and then transform, clip, and render again.

6.5 Screen Transformation

Now that we've covered viewing, projection, and clipping, our final step in transforming our object in preparation for rendering is to map its geometric data from the NDC frame to the screen or device frame. This could represent a mapping to the full display, a window within the display, or an offscreen pixel buffer.

Remember that our coordinates in the NDC frame range from a lower left corner of $(-1, -1)$ to an upper right corner of $(1, 1)$. Real device space coordinates usually range from an upper left corner $(0, 0)$ to a lower right corner (w_s, h_s), where w_s (screen width) and h_s (screen height) are usually not the same. In addition, in screen space the y-axis is commonly flipped so that y values increase as we move down the screen. Some windowing systems allow you to use the standard y direction, but we'll assume the default (Figure 6.27).

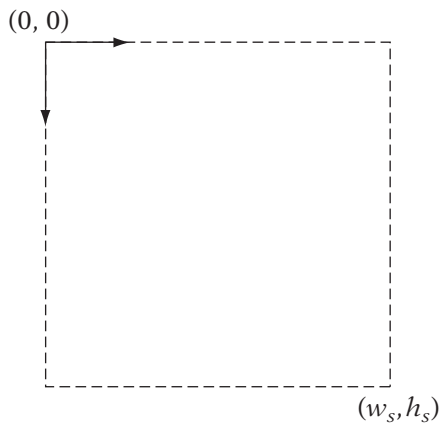


Figure 6.27 View window in standard screen space frame.

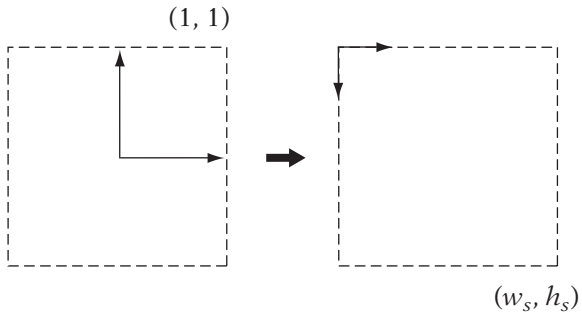


FIGURE 6.28 Mapping NDC space to screen space.

What we'll need to do is map our NDC area to our screen area (Figure 6.28). This consists of scaling it to the same size as the screen, flipping our y direction, and then translating it so that the upper left corner becomes the origin.

Let's begin by considering only the y direction, because it has the special case of the axis flip. The first step is scaling it. The NDC window is two units high, whereas the screen space window is h_s high, so we divide by 2 to scale the NDC window to unit height, and then multiply by h_s to scale to screen height:

$$y' = \frac{h_s}{2} y_{ndc}$$

Since we're still centered around the origin, we can do the axis flip by just negating:

$$y'' = -\frac{h_s}{2} y_{ndc}$$

Finally, we need to translate downwards (which is now the positive y direction) to map the top of the screen to the origin. Since we're already centered on the origin, we need to translate only half the screen height, so

$$y_s = -\frac{h_s}{2} y_{ndc} + \frac{h_s}{2}$$

Another way of thinking of the translation is that we want to map the extreme point $-h_s/2$ to 0, so we need to add $h_s/2$.

A similar process, without the axis flip, gives us our x transformation:

$$x_s = \frac{w_s}{2} x_{ndc} + \frac{w_s}{2}$$

This assumes that we want to cover the entire screen with our view window. In some cases, for example in a split-screen console game, we want to cover only a portion of the screen. Again, we'll have a width and height of our screen space area, w_s and h_s, but now we'll have a different upper left corner position for our area: (s_x, s_y). The first part of the process is the same; we scale the NDC window to our screen space window and flip the y-axis. Now, however, we want to map $(-w_s/2, -h_s/2)$ to (s_x, s_y), instead of $(0, 0)$. The final translation will be $(w_s/2 + s_x, h_s/2 + s_y)$. This gives us our generalized screen transformation in xy as

$$x_s = \frac{w_s}{2} x_{ndc} + \frac{w_s}{2} + s_x \tag{6.5}$$

$$y_s = -\frac{h_s}{2} y_{ndc} + \frac{h_s}{2} + s_y \tag{6.6}$$

Our z coordinate is a special case. As mentioned, we'll want to use z for depth testing, which means that we'd really prefer it to range from 0 to d_s, where d_s is usually 1. This mapping from $[-1, 1]$ to $[0, d_s]$ is

$$z_s = \frac{d_s}{2} z_{ndc} + \frac{d_s}{2} \tag{6.7}$$

We can, of course, express this as a matrix:

$$\mathbf{M}_{ndc \to screen} = \begin{bmatrix} \frac{w_s}{2} & 0 & 0 & \frac{w_s}{2} + s_x \\ 0 & -\frac{h_s}{2} & 0 & \frac{h_s}{2} + s_y \\ 0 & 0 & \frac{d_s}{2} & \frac{d_s}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6.5.1 Pixel Aspect Ratio

Recall that in our projection matrices, we represented the shape of our view window by setting an aspect ratio a. Most of the time it is expected that the value of a chosen in the projection will match the aspect ratio w_s/h_s of the final screen transformation. Otherwise, the resulting image will be distorted. For example, if we use a square aspect ratio ($a = 1.0$) for the projection and a standard aspect ratio of 4:3 for the screen transformation, the image will appear compressed in the y direction. If your image does not quite look right, it is good practice to ensure that these two values are the same.

An exception to this practice arises when your final display has a different aspect ratio than the offscreen buffers that you're using for rendering. For

example, NTSC televisions have 448 scan lines, with 640 analog pixels per scan line, so it is common practice to render to a 640×448 area and then send that to the NTSC converter to be displayed. Using the offscreen buffer size would give an aspect ratio of 10:7. But the actual television screen has a 4:3 aspect ratio, so the resulting image will be distorted, producing stretching in the y direction. The solution is to set $a = 4/3$ despite the aspect ratio of the offscreen buffer. The image in the offscreen buffer will be compressed in the y direction, but then will be proportionally stretched in the y direction when the image is displayed on the television, thereby producing the correct result.

6.6 Picking

Now that we understand the mathematics necessary for transforming an object from world coordinates to screen coordinates, we can consider the opposite case. In our game we may have enemy objects that we'll want to target. The interface we have chosen involves tracking them with our mouse and then clicking on the screen. The problem is: How do we take our click location and use that to detect which object we've selected (if any)? We need a method that takes our 2D screen coordinates and turns them into a form that we can use to detect object intersection in 3D game space. Effectively we are running our pipeline backwards, from the screen transformation to the projection to the viewing transformation (clipping is ignored as we're already within the boundary of our view window).

For the purposes of discussion, we'll assume that we are using the basic OpenGL perspective matrix. Similar derivations can be created using other projections. Figure 6.29 is yet another cross section showing our problem. Once again, we have our view frustum, with our top and bottom clipping planes, our projection plane, and our near and far planes. Point P_s indicates our click location on the projection plane. If we draw a ray (known as a pick ray) from the view position through P_s, we pass through every point that lies underneath our click location. So to determine which object we have clicked on, we need only generate this point on the projection plane, create the specific ray, and then test each object for intersection with the ray. The closest object to the eye will be the object we're seeking.

To generate our point on the projection plane, we'll have to find a method for going backwards from screen space into view space. To do this we'll have to find a means to "invert" our projection. Matrix inversion seems like the solution, but it is not the way to go. The standard projection matrix has zeros in the right-most column, so it's not invertible. But even using the z-depth projection matrix doesn't help us, because (a) the reciprocal divide makes the process nonlinear, and (b) in any case, our click point doesn't have a z value to plug into the inversion.

Instead, we begin by transforming our screen space point (x_s, y_s) to an NDC space point (x_{ndc}, y_{ndc}). Since our NDC to screen space transform is affine, this is easy enough: We need only invert our previous equations 6.5 and 6.6. That gives us

$$x_{ndc} = \frac{2(x_s - s_x)}{w_s} - 1$$
$$y_{ndc} = -\frac{2(y_s - s_y)}{h_s} + 1$$

Now the tricky part. We need to transform our point in the NDC frame to the view frame. We'll begin by computing our z_v value. Looking at Figure 6.29 again, this is straightforward enough. We'll assume that our point lies on the projection plane so the z value is just the z location of the plane or $-d$. This leaves our x and y coordinates to be transformed. Again, since our view region covers a rectangle defined by the range $[-a, a]$ (recall that a is our aspect ratio) in the x direction and the range $[-1, 1]$ in the y direction, we only need to scale to get the final point. The view window in the NDC frame ranges from $[-1, 1]$ in y, so no scale is needed in the y direction and we scale by a in the x direction. Our final screen space to view space equations are

$$x_v = \frac{2a}{w_s}(x_s - s_x) - 1$$
$$y_v = -\frac{2}{h_s}(y_s - s_y) + 1$$
$$z_v = -d$$

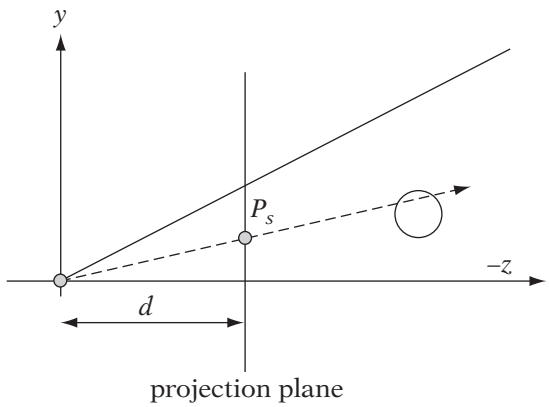


Figure 6.29 Pick ray.

Since this is a system of linear equations, we can express this as a 3×3 matrix:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \begin{bmatrix} \frac{2a}{w_s} & 0 & -\frac{2a}{w_s}s_x - 1 \\ 0 & -\frac{2}{h_s} & \frac{2}{h_s}s_y + 1 \\ 0 & 0 & -d \end{bmatrix} \begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix}$$

From here we have a choice. We can try to detect intersection with an object in the view frame, we can detect in the world frame, or we can detect in the object's local frame. The first involves transforming every object into the view frame and then testing against our pick ray. The second involves transforming our pick ray into the world frame and testing against the world coordinates of each object. For simulation and culling purposes, often we're already pregenerating our world location and bounding information. So, if we're only concerned with testing for intersection against bounding information, it can be more efficient to go with testing in world space. However, usually we test in local space so we can check for intersection within the frame of the stored model vertices. Transforming these vertices into the world frame or the view frame every time we did picking could be prohibitively expensive.

In order to test in the model's local space, we'll have to transform our view space point by the inverse of the viewing transformation. Unlike the perspective transformation, however, this inverse is much easier to compute. Recall that since the view transformation is an affine matrix, we can invert it to get the view-to-world matrix $\mathbf{M}_{view \to world}$. So, multiplying $\mathbf{M}_{view \to world}$ by our click point in the view frame gives us our point in world coordinates:

$$P_w = \mathbf{M}_{view \to world} \cdot P_v$$

We can transform this and our view position E from world coordinates into model coordinates by multiplying by the inverse of the model-to-world matrix:

$$P_l = \mathbf{M}_{world \to model} \cdot P_w$$
$$E_l = \mathbf{M}_{world \to model} \cdot E$$

Then, the formula for our pick ray in model space is

$$R(t) = E_l + t(P_l - E_l)$$

We can now use this ray in combination with our objects to find the particular one the user has clicked on. Chapter 12 discusses how to determine intersection between a ray and an object and other intersection problems.

6.7 Management of Viewing Transformations

Source Code Library: IvEngine
Filename: IvGLHelp

Up to this point we have presented a set of transformations and corresponding matrices without giving some sense of how they would fit into a game engine. While the thrust of this book is not about writing renderers, we can still provide a general sense of how some renderers and application programming interfaces (APIs) manage these matrices, and how to set transformations for a standard API.

The view, projection, and screen transformations change only if the camera is moved. As this happens rarely, these matrices are usually computed once, stored, and then concatenated with the new world transformation every time a new object instance is rendered. How this is handled depends on the API used. The most direct approach is to concatenate the newly set world transform matrix with the others, creating a single transformation all the way from model space to prehomogeneous divide screen space:

$$M_{model \to screen} = M_{ndc \to screen} \cdot M_{projection} \cdot M_{world \to view} \cdot M_{model \to world}$$

Multiplying by this single matrix and then performing three homogeneous divisions per vertex generates the screen coordinates for the object. This is extremely efficient, but ignores any clipping we might need to do. In this case, we can concatenate up to homogeneous space, also known as clip space:

$$M_{model \to clip} = M_{projection} \cdot M_{world \to view} \cdot M_{model \to world}$$

Then we transform our vertices by this matrix, clip against the view frustum, perform the homogeneous divide, and either calculate the screen coordinates using equations 6.5–6.7 or multiply by the NDC to screen matrix, as before.

With more complex renderers, we end up separating the transformations further. For example, OpenGL handles lighting and some clipping prior to projection, so it has separate `GL_MODELVIEW` and `GL_PROJECTION` matrix stacks, to which the appropriate matrices have to be concatenated. The vertices are transformed by the top matrix in the `GL_MODELVIEW` stack, lighting and user-defined clipping are computed, and then the vertices are transformed by the top matrix in the `GL_PROJECTION` matrix. The resulting vertices are clipped in homogeneous space, the reciprocal divide is performed as before, and finally they are transformed to screen space.

In our program, we can set the view and projection matrices in OpenGL by the following code.

```
IvMatrix44 projection, viewTransform;

// compute projection and view transformation
...
```

```
// set in OpenGL
glMatrixMode(GL_PROJECTION);
glLoadMatrix( projection );

glMatrixMode(GL_MODELVIEW);
glLoadMatrix( viewTransform );
```

And when we render an object, concatenating the world matrix can be done by the following code.

```
glMatrixMode(GL_MODELVIEW);

// push copy of view matrix to top of stack
glPushMatrix();

// multiply by world matrix
glMultMatrix( worldTransform );

// render
...

// pop to view matrix
glPopMatrix();
```

The push/pop calls provide a means for storing the view transformation without reloading it into the stack. The call `glPushMatrix()` copies the current matrix — in this case, the view matrix — to a new entry on the top of the stack. The subsequent `glMultMatrix()` will postmultiply the world matrix by the copy of the view matrix at the top of the stack. The resulting local-to-view matrix will be used to transform the vertices of our object. Finally, `glPopMatrix()` removes the current matrix from the top of the stack, restoring the view transformation as the top matrix. The effect is to save the view transformation, multiply by the world transformation and use the result to transform the vertices, and then restore the original view transformation.

Direct3D takes this one step further and manages storage of the view transformation by having three separate matrices: one each for the projective, view, and world transformations. These can be set by using the `IDirect3DDevice*::SetTransform()` method, and any concatenation is handled internally to the API.

This leaves the NDC to screen space transformation. Usually the graphics API will not require a matrix but will perform this operation directly. In the xy directions the user is only expected to provide the dimensions and position of the screen window area, also known as the viewport. In OpenGL this is set by

using the call `glViewport()`. For the z direction, OpenGL provides a function `glDepthRange()`, which maps $[-1, 1]$ to [*near*, *far*], where the defaults for *near* and *far* are 0 and 1, respectively. Similar methods are available for other APIs.

In our case, we have decided not to overly complicate things and are providing simple convenience routines in the IvRenderer class:

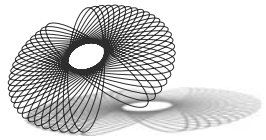
```
IvSetWorldMatrix()
IvSetViewMatrix()
IvSetProjectionMatrix()
IvSetViewport()
```

that act as wrappers for the OpenGL and D3D calls described.

6.8 Chapter Summary

Manipulating objects in the world frame is only useful if we have appropriate techniques for presenting that data. In this chapter we have discussed the viewing, projection, and screen transformations necessary for rendering objects on a screen or image. While we have focused on OpenGL as our rendering API, the same principles apply to Direct3D or any other rendering system. We transform the world to the perspective of a virtual viewer, project it to a view plane, and then scale and translate the result to fit our final display. We also covered how to reverse those transformations to allow one to select an object in view or world space by clicking on the screen. In the following chapters we will discuss how to use the data generated by these transformations to actually set pixels on the screen.

For those who are interested in reading further, most graphics textbooks—such as Möller and Haines [82] and Foley and van Dam [38]—describe the graphics pipeline in great detail. In addition, one of Blinn's collections [8] is almost entirely dedicated to this subject. Various culling techniques are discussed in Möller and Haines [82] as well as Eberly [25]. Finally, the *OpenGL Programming Guide* [85] discusses the particular implementation of the graphics pipeline used in OpenGL.

Chapter 7

Geometry and Programmable Shading

7.1 Introduction

Having discussed in detail in the preceding chapters how to represent, transform, view, and animate geometry, the next three chapters form a sequence that describes the second half of the rendering pipeline. The second half of the rendering pipeline is specifically focused on visual matters: the representation, computation, and usage of color.

This chapter will discuss how we connect the points we have been transforming and projecting to form solid surfaces, as well as the extra information we use to represent the unique appearance of each surface. All visual representations of geometry require the computation of colors; this chapter will discuss the data structures used to store colors and perform basic color computations.

Having shown how to build these renderable surface objects and described the methods of storing and computing colors, we will then lay out the foundations of the rest of the rendering section: the programmable shading and rasterization pipeline. Note that this chapter, unlike the others in the rendering section, is by comparison devoid of pure mathematics. This chapter serves to lay out the fundamental pipeline within which the mathematical work is done: the rendering pipeline. The stages of the framework described in this chapter will be detailed in the later chapters (and to some degree in the previous viewing chapter), where the fascinating mathematical issues

that arise within them can be explored. By its nature, this chapter focuses on the framework itself, the rendering pipeline, and its two most interesting components, the programmable vertex and fragment shader units.

We will also introduce some of the simpler methods of using this programmable pipeline to render colored geometry by introducing the basics of a common high-level shading language, OpenGL's GLSL. Common inputs and outputs to and from the shading pipeline will be discussed, concluding in a detailed introduction to the most complex and powerful of programmable shader source values — image-based texturing. However, this chapter includes only the most basic of programmable shaders, seeking mainly to introduce the rendering pipeline itself.

In Chapter 8, Lighting, we will simultaneously explain the mathematics of real-time light simulation for rendering while demonstrating how to use the programmable shading pipeline to implement dynamic coloring of surfaces. In this chapter we will mix geometric intuitions, the basics of light-related physics, and simulated lighting equations and common approximations thereof with a discussion of more advanced uses of programmable shading.

As the concluding chapter in this sequence, Chapter 9 covers details of the final step in the overall rendering pipeline — rasterization, or the method of determining how to draw the colored surfaces as pixels on the display device. This will complete the discussion of the rendering pipeline.

In each section in these chapters we will relate the basic programming concepts, data structures, and functions that affect the creation, rendering, and coloring of geometry. As we move from geometry representation through shading, lighting, and rasterization, implementation information will become increasingly frequent, as the implementation of the final stages of the rendering pipeline is very much system-dependent. While we will select a particular rendering application programming interface (API) (the book's basic Iv engine) and shading language (OpenGL's GLSL), the basic rendering concepts discussed will apply to most rendering systems.

As a note, we use the phrase *implementation* to refer to the underlying software or "driver" that maps our application calls to a given standard rendering API such as OpenGL or Direct3D into commands for a particular piece of graphics hardware (a graphics processing unit, or GPU, a term coined to recognize the CPU-like rising complexity and performance of modern graphics hardware). OpenGL and Direct3D implementations for a particular piece of graphics hardware are generally supplied with the device by the hardware vendor. A low-level hardware driver is not something that users of these APIs will have to write or even use directly. In fact, the main purpose of OpenGL and other such APIs is to provide a standard interface on top of these widely varying hardware/software three-dimensional (3D) systems. To avoid doubling the amount of implementation-related text in these chapters, most of

the code examples in this and the following rendering chapters will describe the book's `Iv` rendering APIs, supplied as full source code on the book's accompanying CD-ROM. Interested readers may look at the implementations of the referenced `Iv` functions to see how each operation can be written in OpenGL or Direct3D.

7.2 Color Representation

7.2.1 RGB Color Model

To represent color, we will use the additive *RGB* (red, green, blue) color model that is almost universal in real-time 3D systems. Approximating the physiology of the human visual system (which is tuned to perceive color based on three primitives that are close to these red, green, and blue colors), the RGB system is used in all common display devices used by real-time 3D graphics systems. Color cathode ray tubes (or CRTs, such as traditional televisions and computer monitors), flat-panel liquid crystal displays (LCDs), plasma displays, and video projector systems are all based upon the additive RGB system. While some colors cannot be accurately displayed using the RGB model, it does support a very wide range of colors, as proven by the remarkable color range and accuracy of modern television and computer displays. For a detailed discussion of color vision and the basis of the RGB color model, see Malacara [70].

The RGB color model involves mixing different amounts of three predefined *primary* colors of light. These carefully defined primary colors are each named by the colors that most closely match them: red, green, and blue. By mixing independently controlled levels of these three colors of light, a wide range of *brightnesses*, *tones*, and *shades* may be created. In the next few sections we will define much more specifically how we build and represent colors using this method.

7.2.2 Colors as "Vectors"

The levels of each of the three primary colors are independent. In a sense, this is similar to a subset of $\mathbb{R}^3$, but with a "basis" consisting of the red, green, and blue "axes," or components. While these can be thought of as a "basis" for our display device's color space, they are not a basis in any true sense for color in general. The behavior of colors does not always map directly into the concept of a real vector space. However, many of the concepts of real vector spaces are useful in describing color representation and operations.

Our colors will be represented by 3-vectors, with the following basis vectors:

$$(1, 0, 0) \rightarrow \textit{red}$$
$$(0, 1, 0) \rightarrow \textit{green}$$
$$(0, 0, 1) \rightarrow \textit{blue}$$

Often, as a form of shorthand, we will refer to the red component of a color $\mathbf{c}$ as $\mathbf{c}_r$ and to the green and blue components as $\mathbf{c}_g$ and $\mathbf{c}_b$, respectively.

7.2.3 Color Range Limitation

The theoretical RGB color space is semi-infinite in all three axes. There is an absolute zero value for each component, bounding the negative directions, but the positive directions are (theoretically) unbounded. Throughout much of the discussions of coloring, lighting, and shading, we will implicitly assume (or actually declare in the shading language) that the colors are nonnegative real values, potentially represented in the shading system as floating-point numbers.

However, the reality of physical display devices imposes severe limitations on the final output color space. When limited to the colors that can be represented by a specific display device, the RGB color space is not infinite in any direction. Real display devices, such as CRTs (standard "tube" monitors), LCD panel displays, and video projectors all have limits of both brightness and darkness in each color component; these are basic physical limitations of the technologies that these displays use to emit light. For details on the functionality and limitations of display device hardware, Hearn and Baker [54] detail many popular display devices.

Displays have minimum and maximum brightnesses in each of their three color axes, defining the range of colors that they can display. This range is generally known as a display device's *gamut*. The minimum of all color components combine to the device's darkest "black," and the maximum of all color components combine to the device's brightest "white." While it might be possible to create extrema that are not pure black and pure white, these are unlikely to be useful in a general display device.

Every display device is likely to have different exact values for its extrema, so it is convenient to use a standard color space for all devices as sort of "normalized device colors." This color space is built such that

$$(0, 0, 0) \rightarrow \textit{darkest black}$$
$$(1, 1, 1) \rightarrow \textit{brightest white}$$

In the rest of this chapter and the following chapter we will work in these normalized color coordinates. This space defines an RGB "color cube," with black at the origin, white at $(1, 1, 1)$, gray levels down the main diagonal between them (a, a, a), and the other six corners representing pure, maximal red $(1, 0, 0)$, green $(0, 1, 0)$, blue $(0, 0, 1)$, cyan $(0, 1, 1)$, magenta $(1, 0, 1)$, and yellow $(1, 1, 0)$.

The following sections will describe some of the vector operations (and vectorlike operations) we will apply to colors, as well as discussions of how these abstract color vectors map onto their final destinations, namely hardware display devices.

7.2.4 Operations on Colors

Adding RGB colors is done using vector addition; the colors are added componentwise. Adding two colors has the same effect as combining the light from two colored light sources, for example, adding red ($\mathbf{r} = (1, 0, 0)$) and green ($\mathbf{g} = (0, 1, 0)$) gives yellow:

$$\mathbf{r} + \mathbf{g} = (1, 0, 0) + (0, 1, 0) = (1, 1, 0)$$

The operation of adding colors will be used through our lighting computations to represent the addition of light from multiple light sources and to add the multiple forms of light that each source can apply to a surface.

Scalar multiplication of RGB colors ($s\mathbf{c}$) is computed in the same way as with vectors, multiplying the scalar times each component, and is ubiquitous in lighting and other color computations. It has the result of increasing ($s > 1.0$) or decreasing ($s < 1.0$) the luminance of the color by the amount of the scalar factor. Scalar multiplication is most frequently used to represent light attenuation due to various physical and geometric lighting properties.

One important vector operation that is used somewhat rarely with colors is vector length. While it might seem that vector length would be an excellent (if expensive) way to compute the luminance of a color, the nature of human color perception does not match the Euclidean norm of the linear RGB color space. Luminance is a "norm" that is affected by human physiology. The human eye is most sensitive to green, less to red, and least sensitive to blue. As a result, the equal weighting given to all components by the Euclidean norm means that blue contributes to the Euclidean norm far more than it contributes to luminance.

Although there are numerous methods used to compute the luminance of RGB colors as displayed on a screen, a common method for modern CRT screens (assuming nonnegative color components) is

$$luminance(\mathbf{c}) = 0.2125\,\mathbf{c}_r + 0.7154\,\mathbf{c}_g + 0.0721\,\mathbf{c}_b$$

Or basically, the dot product of the color with a "luminance reference color." The three color-space transformation coefficients used to scale the color components are basically constant for modern, standard CRT screens but do not necessarily apply to television screens, which use a different set of luminance conversions. Discussion of these may be found in Poynton [94]. Note that luminance is *not* equivalent to perceived brightness. The luminance as we've computed it is linear with respect to the source linear RGB values. Brightness as perceived by the human visual system is nonlinear and subject to the overall brightness of the viewing environment, as well as the viewer's adaptation to it. See Cornsweet [20] for a related discussion of the physiology of human visual perception.

An operation that is rarely applied to geometric vectors but is used very frequently with colors is componentwise multiplication. Componentwise multiplication takes two colors as operands and produces another color as its result. We will represent the operation of componentwise multiplication of colors as "$\cdot$," or in shorthand by placing the colors next to one another (as we would multiply scalars), and the operation is defined as follows:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{ab} = (\mathbf{a}_r\mathbf{b}_r, \mathbf{a}_g\mathbf{b}_g, \mathbf{a}_b\mathbf{b}_b)$$

This operation is often used to represent the *filtering* of one color of light through an object of another color. In such a situation, one operand is assumed to be the light color, while the other operand is assumed to be the amount of light of each component that is passed by the filter. Another use of componentwise color multiplication is to represent the reflection of light from a surface — one color represents the incoming light and the other represents the amount of each component that the given surface reflects (the surface's reflectivity). We will use this frequently in Chapter 8 when computing lighting. For example, a color $\mathbf{c}$ and a filter (or surface) $\mathbf{f} = (1, 0, 0)$, results in

$$\mathbf{cf} = (\mathbf{c}_r, 0, 0)$$

or the equivalent of a pure red filter; only the red component of the light was passed, while all other light was blocked. This operation will be used constantly in color lighting computations.

7.2.5 Alpha Values

Frequently, RGB colors are augmented with a fourth component, called *alpha*. Such colors are often written as *RGBA* colors. Unlike the other three components, the alpha component does not represent a specific color basis, but

rather defines how the combined color interacts with other colors. The most frequent use of the alpha component is an opacity value, which defines how much of the surface's color is controlled by the surface itself and how much is controlled by the colors of objects that are behind the given surface. When alpha is at its maximum (we will define this as 1.0), then the color of the surface is independent of any objects behind it. The red, green, and blue components of the surface color may be used directly, for example, in representing a solid concrete wall. At its minimum (0.0), the RGB color of the surface is ignored and the object is invisible, as with a pane of clear glass for instance. At an intermediate alpha value, such as 0.5, the colors of the two objects are blended together; in the case of alpha equaling 0.5, the resulting color will be the componentwise average of the colors of the surface and the object behind the surface.

For the most part, alpha will be treated like any other color component until rasterization. We will discuss the uses of the alpha value (known as *alpha blending*) in Chapter 9 on rasterization. In a few cases, rendering APIs handle alpha a little differently from other color components (mention will be made of these situations as needed).

Remapping Colors into the Unit Cube

Although devices cannot display colors outside of the range defined by their $(0, 0, 0) \ldots (1, 1, 1)$ cube, colors outside of this cube are often seen during intermediate color computations such as lighting. In fact, the very nature of lighting can lead to final colors with components outside of the $(1, 1, 1)$ limit. During lighting computations, these are generally allowed, but prior to assigning final colors to the screen, all colors must be within the normalized cube. This requires either the hardware, the device driver software, or the application to somehow remap or limit the values of colors so that they fall within the unit cube.

The simplest and easiest method is to clamp the color on a per-component basis:

$$safe(\mathbf{c}) = (clamp(\mathbf{c}_r), clamp(\mathbf{c}_g), clamp(\mathbf{c}_b))$$

where

$$clamp(x) = max(min(x, 1.0), 0.0)$$

However, it should be noted that such an operation can cause significant perceptual changes to the color. For example, the color $(1.0, 1.0, 10.0)$ is predominantly blue, but its clamped version is pure white $(1.0, 1.0, 1.0)$. In general, clamping a color can lead to the color becoming less *saturated,* or

less colorful. While this might seem unsatisfactory, it actually can be beneficial in some forms of simulated lighting, as it tends to make overly bright objects appear to "wash out," an effect that can perceptually appear rather natural under the right circumstances.

Another, more computationally expensive method is to rescale all three color components of any color with a component greater than 1.0 such that the maximal component is 1.0. This may be written as

$$safe(\mathbf{c}) = \frac{(max(\mathbf{c}_r, 0), max(\mathbf{c}_g, 0), max(\mathbf{c}_b, 0))}{max(\mathbf{c}_r, \mathbf{c}_g, \mathbf{c}_b, 1)}$$

Note the appearance of 1 in the *max* function in the denominator to ensure that colors already in the unit cube will not change—it will never increase the color components. While this method does tend to avoid changing the overall saturation of the color, it can produce some unexpected results. The most common issue is that extremely bright colors that are scaled back into range can actually end up appearing darker than colors that did not require scaling. For example, comparing the two colors $\mathbf{a} = (1, 1, 0)$ and $\mathbf{b} = (10, 5, 0)$, we find that after scaling, $\mathbf{b} = (1, 0.5, 0)$, which is significantly darker than $\mathbf{a}$.

Scaling works best when it is applied equally (or at least coherently) to all colors in a scene, not to each color individually. There are numerous methods for this, but one such method involves finding the maximum color component of any object in the scene, and scaling all colors equally such that this maximum maps to 1.0. This is somewhat similar to a camera's auto-exposure system. By scaling the entire scene by a single scalar, color ratios between objects in the scene are preserved. Figure 7.1 shows two different color-range limitation methods for the same source image. In Figure 7.1(a), we clamp the values that are too large to display. Note that this results in a loss of image detail in the brightest sections of the image, which become pure white. In Figure 7.1(b), we rescale all of the colors in the image based on the maximum value method described above. The details in the brightest areas of the screen are retained. However, even this method is not perfect. The rescaling of the colors does sacrifice some detail in the darker shadows of the image.

A more advanced method known generally as *tone mapping* remaps regions of an image differently; a very bright section of the scene may be darkened to fit the range (e.g., a bright, cloud-streaked sky), while the shadowed sections of the image actually may be scaled to be brighter so that details are not lost in the shadows. The scaling may be different for different sections of the image, but the remapping is done in a regionally coherent method so that the relative brightness of related objects are reasonable. Regionally coherent means that we take the brightness of the region surrounding any point on the screen and try to keep the relative bright–dark relationships. A common trick in a daytime image of buildings and sky would be to darken the sky to fit in

FIGURE 7.1 Color-range limitation methods: (a) image colors clamped, and (b) image colors rescaled.

range and brighten the buildings to be less in shadow. While we are applying different scalings to different parts of the image (darkening to the sky and brightening to the buildings), the relative brightnesses within the buildings' region of the image are kept intact, and the relative brightnesses within the sky's regions of the image are kept intact. Thus, the sky and the buildings each look like what we'd expect, but the overall image fits within the limited brightness range.

These techniques are often used in high dynamic range (HDR) rendering, in which wide orders of magnitude exist in the computed lighting, but are then mapped down to the unit cube in a manner that forms a vibrant image. Figure 7.2 shows the same image for Figure 7.1, but tone-mapped to retain details in both the shadows and highlights. The shadowed and highlighted areas are processed independently to avoid losing detail in either.

HDR rendering is growing in popularity in 3D games and other applications as GPU feature sets and performance have improved. Many examples of HDR rendering may be found at the developers' websites of the major GPU vendors [1, 84].

Figure 7.2 A tonemapped image.

7.2.6 Color Storage Formats

A wide range of color storage formats are used by modern rendering systems, both floating point and fixed point (as well as one or two hybrid formats). Common RGBA color formats include:

- Single-precision floating-point components (128 bits for RGBA color).
- Half-precision floating-point components (64 bits for RGBA color).
- 16-bit unsigned integer components (64 bits for RGBA color).
- 8-bit unsigned integer components (32 bits for RGBA color).
- Shared exponent extended-range formats. In the most common of these formats, red, green, and blue represent 0-dot-8 fixed-point mantissas, while a final 8-bit shared exponent is used to scale all three components. This is not as flexible as a floating-point value per color component (since all components share a single exponent), but it can represent a huge dynamic range of colors using only 32 bits for an RGB color.

In general, the floating-point formats are used as would be expected (in fact, on modern systems, the single-precision floating-point colors are now IEEE 754 compliant, making them useful for noncolor computations as well). However, the integer formats have a special mapping in most graphics systems. An integer value of zero maps to zero, but the maximal value maps to 1.0. Thus, the integer formats are slightly different than those seen in any fixed-point format.

While a wide range of color formats are available to applications, a small subset of them cover most use cases. Internal to the programmable rendering pipeline, floating-point values are the most popular intermediate result formats. However, floating-point values are *not* the most popular format for shading output, the values that are stored in the frame buffer or other image buffer. Perhaps the most popular format for final color storage is unsigned 8-bit values per component, leading to 3 bytes per RGB color, a system known as *24-bit color*, or in some cases, by the misnomer "true color." With an alpha value, the format becomes 32 bits per pixel, which aligns well on modern 32- and 64-bit CPU architectures. Another common format is to use 5 bits each for red and blue and 6 bits for green, a format that requires 16 bits per pixel. This system, which sometimes goes by the name *high color*, is interesting in that it includes different amounts of precision for green than for red or blue. As we've discussed, the human eye is most sensitive to green, so the additional bit in the 16-bit format is assigned to it. However, the number of pure gray values in this format is still $2^5 = 32$, since the additional bit of precision in green must be zero for all grays (or else the system risks having some slightly green-tinted gray values).

The historical reasons for using these lower-precision formats are storage space requirements, computational expense, and the fact that display devices often have the ability to display only 5–8 bits of precision per component. Even 32 bits per pixel requires one-quarter the amount of storage that is needed for floating-point RGBA values. Using full floating-point numbers for output colors (the colors that are drawn to the output LCD or CRT screen) is actually overkill, due to the limitations of current display device color resolution. For example, current CRTs and LCD displays have dynamic ranges (the ratio of luminance between the brightest and darkest levels that can be displayed by the devices) of between 200:1 and 500:1. These ratios mean that current display devices cannot deliver anywhere near the eye's full range of perceived brightness or darkness. There are display technologies that can represent more than 24-bit color, but these are still the exception, rather than the rule. As these display devices become more common, device-level color representations will require more bits per component in order to avoid wasting the added precision available from these new displays.

Research has shown that the human visual system (depending on lighting conditions, etc.) can perceive between 1 million and 7 million colors, which leads to the (erroneous) theory that 24-bit color display systems, with their

$2^{24} \approx 16.7$ million colors, are more than sufficient. While it is true that the number of different color "names" in a 24-bit system (where a color is "named" by its 24-bit RGB triple) is a greater number than the human visual system can discern, this does not take into account the fact that the colors being generated on current display devices do not map directly to the 1–7 million colors that can be discerned by the human visual system. Current display devices cannot display the entire range of colors that the human eye can discern. In addition, in some color ranges, different 24-bit color "names" appear the same to the human visual system (the colors are closer to one another than the human eye's *just noticeable difference*, or JND). In other words, 24-bit color wastes precision in some ranges, while lacking sufficient precision in others. Current 24-bit "true color" display systems are not sufficient to cover the entire range of human vision, either in range or in precision. Having said this, current display devices are still quite convincing to the human eye and will continue to improve.

7.3 Points and Vertices

So far, we have discussed points as our sole geometry representation. As we begin to abstract to the higher level of a surface, points will become insufficient for representing the attributes of an object or for that matter the object itself. The first step in the move toward a way of defining an object's surface is to associate additional data with each point. Combined together (often into a single data structure), each point and its additional information form what is often called a *vertex*. In a sense, a vertex is a "heavy point": a point with additional information that defines some properties of the surface around it.

7.3.1 Per-Vertex Attributes

Within a vertex, the most basic value is the position of the vertex, generally a 3D point that we will refer to as P_V in later sections.

Other than vertex position, perhaps the most basic of the "standard" vertex attributes are colors. Common additions to a vertex data structure, vertex colors are used in many different ways when drawing geometry. Much of the remainder of this chapter will discuss the various ways that per-vertex colors can be assigned to geometry, as well as the different ways that these vertex colors are used to draw geometry to the screen. We will generally refer to the vertex color as C_V (and will sometimes specifically refer to the vertex alpha as A_V, even though it is technically a component of the overall color).

Another data element that can add useful information to a vertex is a vertex normal. This is a unit-length 3-vector that defines the "orientation" of

the surface in an infinitely small neighborhood of the vertex. If we assume that the surface passing through the vertex is locally planar (at least in an infinitely small neighborhood of the vertex), the surface normal is the normal vector to this plane (recall the discussion of plane normal vectors from Chapter 2). In most cases, this vector is defined in the same space as the vertices, generally model (a.k.a. object) space. As will be seen later, the normal vector is a pivotal component in lighting computations. We will generally refer to the normal as $\hat{\mathbf{n}}_V$.

A vertex attribute that we will use frequently later in this chapter is a texture coordinate. This will be discussed in detail in the sections in this chapter on texturing and in parts of the following two chapters; basically, a set of texture coordinates is a real-valued 2-vector (most frequently, although they also may be scalars or 3-vectors) per vertex that defines the position of the vertex within a smooth parameterization of the overall surface. These are used to map two-dimensional (2D) images onto the surface in a shading process known as texturing. A vertex may have more than one set of texture coordinates, representing the mapping of the vertex in several different parameterizations.

Finally, owing to the general and extensible nature of programmable shading, an object's vertices may have other sets of per-vertex attributes. Most common are additional values similar to the ones listed above; per-vertex color values, per-vertex directional vectors of some sort, or per-vertex texture coordinates. However, other programmable shaders could require a wealth of different vertex attributes; most shading systems support scalar vertex attributes as well as generic 2D, 3D, and 4D vectors. The meaning of these vectors are dependent upon the shading program itself.

7.3.2 An Object's Vertices

For any geometric object, its set of vertices can be represented as an array of structures. Each array element contains the value for each of the vertex attributes supported by the object. Note that for a given object, all of the vertices in the array have the same type of structure. If one vertex has a particular attribute, they all will contain that attribute (likely with a different value). An example of the vertex structure for an object with position values, a color, and one set of texture coordinates is shown below.

```
struct IvTCPVertex
{
   IvVector2 texturecoord;
   IvColor color;
   IvVector3 position;
};
```

A smaller, simpler vertex with just position and normal might be as follows:

```
struct IvNPVertex
{
   IvVector3 normal;
   IvVector3 position;
};
```

Along with the C or C++ representation of a vertex, an application must be able to communicate to the rendering API how the vertices are laid out. Each rendering API uses its own system, but two different methods are common; the simpler (but less flexible) method is for the API to expose some fixed set of supported vertex formats explicitly and use an enumerated type label to represent each of these formats. All of an application's geometry must be formatted to fit within the fixed set of supported vertex formats in this case. The more general system is for an API to allow the application to specify the type (float, etc.); usage (position, color, etc.); dimension (1D, 2D, etc.); and stride (bytes between the attribute for one vertex and the next) of each active attribute. This system is far more flexible, but can greatly increase the complexity of the rendering API implementation. The latter is common in modern graphics APIs, such as Direct3D's DX9 and OpenGL. The former method is used in `Iv` for the purposes of simplicity and ease of cross-platform support. `Iv` uses the following enumeration to define the vertex formats it supports:

```
enum IvVertexFormat
{
   kCPFormat,      // color, position
   kNPFormat,      // normal, position
   kTCPFormat,     // texture coord, color, position
   kCNPFormat,     // color, normal, position
   kTNPFormat      // texture coord, normal, position
};
```

This enumeration is used in various places in the `Iv` rendering engine to declare the format of a given vertex or array of vertices to the system.

Some rendering APIs allow for the vertex attributes to be "non-interleaved"; that is, the application keeps independent packed arrays of each vertex attribute. This so-called "structure of arrays" format is generally less popular in modern APIs, as the interleaved formats provide better cache coherence — in an interleaved format, accessing one attribute in a vertex is likely to load the entire vertex into cache. There is one notable exception: If

some of an object's vertex attributes are computed on the host CPU, it may make sense to keep them in their own array, while leaving the constant vertex attributes in another fully interleaved vertex array. This allows the dynamic data to be modified without touching or retransferring the static data to device memory. We will assume an interleaved vertex format for the remainder of the rendering discussions.

Vertex Buffers

Programmable shaders and graphics rendering pipelines implemented entirely in dedicated hardware have made it increasingly important for as much rendering-related data as possible to be available to the GPU in device-local memory, rather than system memory. Modern graphics APIs all include the concept of a *vertex buffer* or *vertex buffer object*, an opaque handle that represents source vertex data resident in GPU memory.

In order to use vertex buffers to render an object, an application must make calls to the rendering API to allocate enough storage for the object's array of vertices in GPU memory. Then, some method is used to transfer the vertex array from system memory to GPU memory. Having transferred the data, the application can then use the opaque handle to render the geometry at peak performance. Note that once vertex array data are in GPU memory, it is usually computationally expensive to modify them. Thus, vertex buffers are most frequently used for data that the CPU does not need to modify on a per-frame basis. Over time, as programmable shaders have become more and more powerful, there have been fewer and fewer (if any) per-vertex operations that need to be done on the CPU, thus making it more easily possible to put all vertex data in static vertex buffers.

A common vertex buffer creation sequence in many APIs is to create the vertex buffer, passing in the vertex format and number of vertices, but no data. The resulting vertex buffer is then "locked," which returns a system memory pointer that can be filled with vertex array data. Finally, the buffer is "unlocked," which releases access to the system memory pointer and (if needed) transfers the vertex data to GPU-accessible memory. In `Iv`, the sequence is as follows:

```
IvResourceManager& manager;
// ...

   // Create a vertex buffer with 1024 vertices
   // Each vertex has a color and position
   IvVertexBuffer* buffer
      = manager.CreateVertexBuffer(kCPFormat, 1024);
```

```
// Lock the vertex buffer and cast to the correct
// vertex format
IvCPVertex* verts
   = (IvCPVertex*)buffer->BeginLoadData();

// Loop over all 1024 vertices in verts and
// fill in the data...
// ...
// Unlock the buffer, so it can be used
buffer->EndLoadData();
```

The vertex buffer is now filled with data and ready to be used to render.

7.4 Surface Representation

In this section we will discuss another important concept used to represent and render objects in real-time 3D graphics: the concept of a surface and the most common representation of surfaces in interactive 3D systems, sets of triangles. These concepts will allow us to build realistic-looking objects from the sets of vertices that we have discussed thus far.

In Chapter 2 we introduced the concept of a triangle, a subset of a plane defined by the convex combination of three noncollinear points. In this chapter we will build upon this foundation and make frequent use of triangles, the normal vector to a triangle, and barycentric coordinates. A quick review of the sections of Chapter 2 covering these topics is recommended.

While most of the remainder of this chapter focuses only on the assignment of colors to objects for the purposes of rendering, the object and surface representations we will discuss are useful for far more than just rendering. Collision detection, picking, and even artificial intelligence all make use of these representations.

7.4.1 Vertices and Surface Ambiguity

Unstructured collections of vertices (sometimes called *point clouds*) generally cannot represent a surface unambiguously. For example, draw a set of ten or so dots representing points on a piece of paper. There are numerous ways one could connect these 2D points into a closed curve (a 1D "surface") or even into several smaller curves. This is true even if the vertices include normal vectors, as these normal vectors only define the orientation of the surface in an infinitely small neighborhood of the vertex. Without additional structure, either implicit or explicit, a finite set of points rarely defines an unambiguous surface.

A cloud of points that is infinitely dense on the desired surface can represent that surface. Obviously, such a directly stored collection of unstructured points would be far too large to render in real time (or even store) on a computer. We need a method of representing an infinitely dense surface of points that requires only a finite amount of representational data.

There are numerous methods of representing surfaces, depending on the intended use. Our requirements are that we can make direct use of the conveniently defined vertices that our geometry pipeline generates, and that the representation we use is efficient to render. As it turns out, we have already been introduced to such a representation in one of the earliest sections of the book: planar triangles.

7.4.2 Triangles

The most common method used to represent 3D surfaces in real-time graphics systems is simple, scalable, requires little additional information beyond the existing vertices, and allows for direct rendering algorithms; it is the approximation of surfaces with triangles, or *tessellation*. Tessellation refers not only to the process that generates a set of triangles from a surface but also to the triangles and vertices that result.

Triangles, each represented and defined by only three points (vertices) on the surface, are connected point to point and edge to edge to create a piecewise flat ("faceted") approximation of the surface. By varying the number and density of the vertices (and thus the triangles) used to represent a surface, an application may make any desired trade-off between compactness/rendering speed and accuracy of representation. Representing a surface with more and more vertices and triangles will result in smaller triangles and a smoother surface, but will add rendering expense and storage overhead owing to the increased amount of data representing the surface.

One concept that we will use frequently with triangles is that of barycentric coordinates. From the discussion in Chapter 2, we know that any point in a triangle may be represented by an element of $\mathbb{R}^2$ (s, t) such that $0.0 \leq s, t \leq 1.0$. These coordinates uniquely define each point on a nondegenerate triangle (i.e., a triangle with nonzero area). We will often use barycentric coordinates as the domain when mapping functions defined across triangles, such as color.

7.4.3 Connecting Vertices into Triangles

To create a surface representation from the set of vertices on the surface, we will simply "connect the dots." That is, we will generate additional information

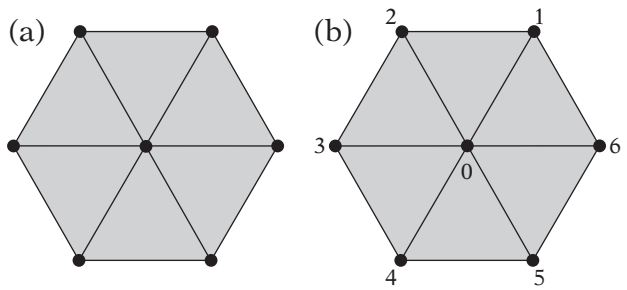


FIGURE 7.3 A hexagonal configuration of triangles: (a) configuration, (b) seven shared vertices, and (c) index list for shared vertices.

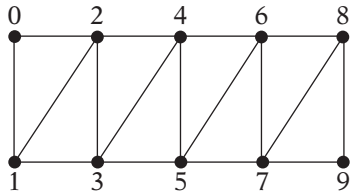
for rendering that joins sets of three vertices by spanning them with a triangle. As an example, Figure 7.3(a) depicts a fan-shaped arrangement of six triangles (defining a hexagon) that meet in a single point. The vertex array for this geometry is an array of seven vertices; six around the edge and one in the center. Figure 7.3(b) shows these seven vertices, numbered with their array indices in the vertex array. However, this array alone does not define any information about the triangles in the object.

Indexed geometry, or indexed triangle lists, bridge this gap. It defines an object with two arrays: the vertex array we have already discussed, and a second array of integral values for the triangle connectivities, called the index (or element) array. The *index array* is an array of integers that represent indices (offsets) into the vertex array; there are three times as many indices in the index array as there are triangles in the object. Each set of three adjacent indices represents a triangle. The indices are used to look up vertices in the vertex array; the three vertices are joined into a triangle. Figure 7.3(c) shows the index list for the hexagon example.

Note the several benefits of indexed geometry. First, vertices can be reused in as many triangles as desired simply by using the same index value several times in the index array. This is shown clearly by the hexagon example. One of the vertices (the central vertex) appears in every single triangle! If we had to duplicate a vertex each time it was used in a triangle, the memory requirements would be much higher, since even small vertex structures take more space than an index value. Index values are generally 16- or 32-bit unsigned integers. A 16-bit index value can represent a surface made up of up to 65,536 vertices, more than enough for the objects in many applications, while a

32-bit index array can represent a surface with more than 4 billion vertices (essentially unlimited).

Most rendering APIs support a wide range of indexed geometry. Indexed triangle lists, such as the ones we've just introduced, are simple to understand but are not as optimal as other representations. The most popular of these more optimal representations are *triangle strips*, or *tristrips*. In a triangle strip, the first three vertex indices represent a triangle, just as they do in a triangle list. However, in a triangle strip, each additional vertex (the fourth, fifth, etc.) generates another triangle — each index generates a triangle out of itself and the two indices that preceded it (e.g., 0-1-2, 1-2-3, 2-3-4, …). This forms a ladderlike strip of triangles (note that each triangle is assumed to have the reverse orientation of the previous triangle — counterclockwise, then clockwise, then counterclockwise again, etc.). Then, too, whereas triangle lists require $3T$ indices to generate T triangles, triangle strips require only $T+2$ indices to generate T triangles. An example of the difference between the size of index arrays for triangle lists and triangle strips is shown in Figure 7.4. Much research has gone into generating optimal strips by maximizing the number of triangles while minimizing the number of strips, since there is a two-vertex "overhead" to generate the first triangle in a strip. The longer the strip, the lower the average number of indices required per strip. Most consumer 3D hardware that is available today renders triangle strips at peak performance, because each new triangle reuses two previous vertices, requiring only one new vertex (and in the case of indexed primitives, one new index) per triangle. This minimizes transform work on the GPU, as well as potential "traffic" over the bus that connects the CPU to the GPU.



Index array for triangle list:
0,1,2, 1,3,2, 2,3,4, 3,5,4, 4,5,6, 5,7,6, 6,7,8, 7,9,8
(24 indices)

Index array for triangle strip:
0,1,2,3,4,5,6,7,8,9
(10 indices)

FIGURE 7.4 The same object as a triangle list and a triangle strip.

Indexed rendering is not the only way to render triangle lists, strips, etc. The other common method is nonindexed geometry, and is equivalent to dereferencing the index list into an array of vertex structures. In other words, a nonindexed triangle list with T triangles would use no index list, but would use a vertex array with $3T$ vertices. Any vertices that were shared in the indexed case must be duplicated in the nonindexed case. This is generally suboptimal, since there is no vertex reuse. In this book we will discuss only indexed geometry.

Index Buffers

Most GPUs can link vertices and indices into triangles without any CPU intervention. Thus, it is useful to be able to place index arrays into GPU-accessible memory. These objects are called *index buffers*, and they are directly analogous to the vertex buffers discussed previously. The only difference is that the format of an index buffer is far more limited; in `Iv`, only 32-bit indices are supported and are assumed. `Iv` code to create and fill an index buffer is shown below.

```
IvResourceManager& manager;
// ...
   // Create an index buffer with 999 indices
   // With triangle lists, this would be 333 triangles
   IvIndexBuffer* buffer = manager.CreateIndexBuffer(999);

   // Lock the index buffer and cast to the correct
   // index format
   unsigned int* indices
      = (unsigned int*)buffer->BeginLoadData();

   // Loop over all 999 indices and fill in the data...
   // ...
   // Unlock the buffer, so it can be used
   buffer->EndLoadData();
```

7.4.4 Drawing Geometry

The final step toward rendering geometry from an application point of view is to pass the required information into the rendering API to initiate the draw operation. Submitting geometry to the rendering API generally takes the form of a draw call. APIs differ on which subset of the geometry information is passed to the draw call and which is set as the current state beforehand, but the basic pieces of information that define the inputs to the draw call include

at least the array of vertices, array of indices, type of primitive (list, strip, etc.), and rendering state defining the appearance of the object. Some APIs may also require the application to specify the location of each component (normal, position, etc.) within the vertex structure. The `Iv` rendering engine sets up the geometry and connectivity, and renders in a single call, as follows:

```
IvRenderer& renderer;
IvVertexBuffer* vertexBuffer;
IvIndexBuffer* indexBuffer;
// ...

   renderer.Draw(kTriangleListPrim, vertexBuffer, indexBuffer);
```

Note the enumerated type used to specify the primitive. In this case, we are drawing an indexed triangle list (`kTriangleListPrim`), but we could have specified a triangle strip (`kTriangleStripPrim`) or other primitive as listed in `IvPrimType`, assuming that the index data were valid for that type of primitive (each primitive type uses its index list a little differently, as discussed previously).

Once the geometry is submitted for rendering, the work really begins for the implementation and 3D hardware itself. The implementation passes the object geometry through the rendering pipeline and finally (if the geometry is visible) onto the screen. The following sections will detail the most common structure of the rendering pipeline in modern graphics APIs.

7.5 Rendering Pipeline

The basic rendering pipeline is shown in Figure 7.5. The flow is quite simple and will be the basis for much of the discussion in this chapter. Some of the items in the diagram will not yet be familiar. In the remainder of this chapter we will fill in these details. The flows are as follows:

1. **Primitive Processing.** The pipeline starts with the triangle indices, which determine on a triangle-by-triangle basis which vertices in the array are required to define each triangle.
2. **Per-Vertex Operations.** All required vertices (which contain surface positions in model space along with the additional vertex attributes) are processed as follows:

   (a) The positions are transformed into homogeneous space using the model view and projection matrices.

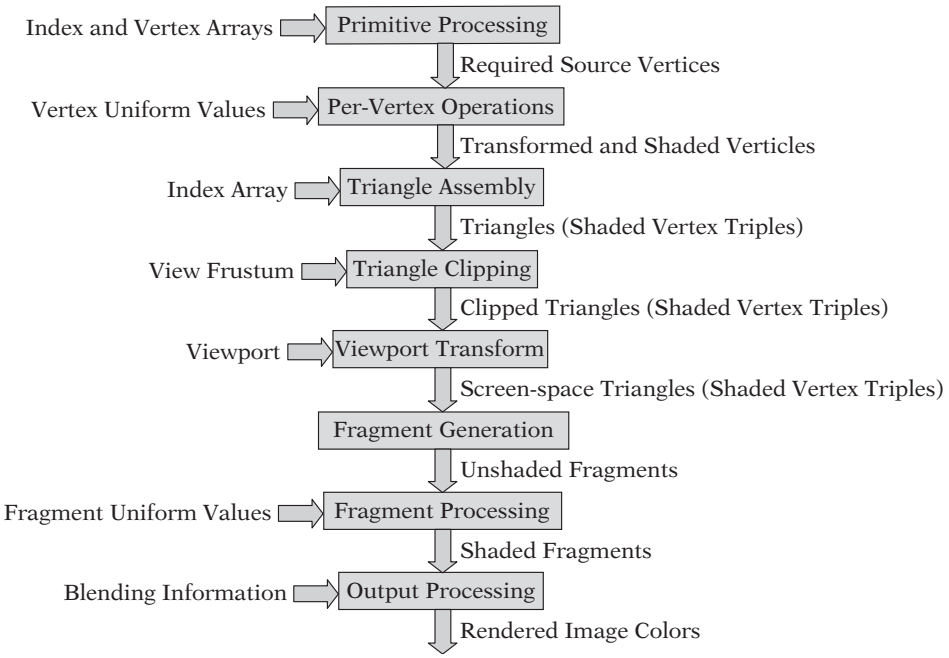


FIGURE 7.5 Details of the rendering pipeline.

(b) Additional per-vertex items such as lit vertex colors are computed based on the positions, normals, etc.

3. **Triangle Assembly.** The transformed vertices are grouped into triples representing the triangles to be rendered.
4. **Triangle Clipping.** Each homogeneous-space triangle is clipped and/or culled as required to fall within the view rectangle.
5. **Viewport Transform.** The resulting clipped triangles are transformed into screen space.
6. **Fragment Generation.** Triangles are "sampled," generating pixel-aligned samples, called fragments.
7. **Fragment Processing.** The final color and other properties of the surface are computed for each fragment.
8. **Output Processing.** The final fragments are combined with those from other objects that are a part of the scene to generate the final rendered image.

The rendering section of this book covers all of these steps in various levels of detail. In this chapter we have already discussed the basics of indexed triangle primitives (primitive processing and triangle assembly). In Chapter 6 we discussed projection of vertices (per-vertex operations), clipping and culling (triangle clipping), and transformation into screen space (viewport transform). In this chapter we will provide an overview of other per-vertex operations and fragment processing. In Chapter 8, Lighting, we will provide details on how light–surface interaction can be simulated in per-vertex operations and fragment processing. Finally, the details of how fragments are generated and processed (fragment generation and processing), as well as how they are output to the device (output processing), are discussed in Chapter 9, Rasterization.

7.5.1 Fixed-Function versus Programmable Pipelines

The above pipeline has been common to rendering systems and APIs for over a decade. Initially, the major rendering APIs such as OpenGL 1.x (and OpenGL ES 1.x) and Direct3D's DX3 through DX7 implemented each stage with basically fixed functionality, modified only by a limited number of settings and switches. As features multiplied in commercial 3D systems, the switches and settings became more and more complex and often began to interact in confusing ways. The APIs became bloated and complicated, even though they were still unable to represent the full flexibility of the new hardware.

As a result, starting with APIs like OpenGL 2.0 and Direct3D's DX8, graphics systems have added flexibility. While the classic fixed-function pipelines were still available to applications, the APIs included new interfaces that allowed several of the most important fixed-function stages to be replaced with application-provided "shader" code. The major stages that were replaced with programmability were the per-vertex operations and fragment processing. Rather than use a growing number of prespecified switches and controls, these APIs added programmable shaders, which replaced the fixed-function stages with application-supplied simple programs that turned the inputs of the stages into the desired application outputs quite directly. In fact, Direct3D's DX10 and the mobile 3D API OpenGL ES 2.0 (along with other APIs of that generation) eschew the fixed-function pipeline entirely; *only* shaders are supported.

While each API used its own programming languages for these shaders, they all progressed in similar manners. The initial shading languages were similar to CPU assembly code: low-level instructions requiring the

programmer to assign inputs, outputs, and temporaries to a limited set of available registers. These were difficult to program and often included confusing limitations. However, as the 3D rendering hardware became more capable, the register sets and instructions became more powerful and general. This led to the true real-time shading revolution.

Hardware vendors and graphics API vendors began to design and standardize high-level shading languages. The three major high-level shading languages used for interactive 3D graphics are NVIDIA's Cg (C for graphics) [35], Microsoft's HLSL (high-level shading language), and OpenGL's GLSL (GL shading language) [99]. While each of these languages has significant differences, they are all remarkably similar. They all have the basic feel of C or C++, and thus switching between them is generally quite easy. Since OpenGL's GLSL is widely available, is supported by both OpenGL 2.0 and OpenGL ES 2.0 (the latter with some limitations, known as GLSL-E), and is quite clean, we will use it exclusively for in-text shading language examples. However, the other shading languages are capable of the same operations in relatively similar ways.

The remainder of this book will deal exclusively with shader-based pipelines. For the examples we will use, shaders are more illustrative and simpler. As we shall see in the lighting chapter (Chapter 8), high-level shading languages make it possible to directly translate shading and lighting equations into shader code. This is the additional value of shaders; while they make complex effects possible, they also make simple shading equations quite efficient by avoiding all of the conditionals and flag-checking required by a fixed-function pipeline's settings.

7.6 Shaders

7.6.1 Using Shaders to Move from Vertex to Triangle to Fragment

Vertex shaders (VS) and fragment shaders (FS, also known in some APIs as pixel shaders) are, at their core function, very similar. They each take input values that represent a single entity, and output values that define additional properties of that entity. In the case of a vertex shader, the entity in question is a vertex, or source surface position and additional attributes as discussed previously in this chapter. In the case of a fragment shader, the entity is a "fragment" or sample representing an infinitesimally small region of the surface being rendered. In Chapter 9 on rasterization, we will see that there is actually

a much more precise definition of a fragment, but for now, the basic concept is that it is a sample somewhere on the surface of the object, generally at a point in the interior of one of the triangles, not coincident with any single vertex defining the surface.

The "one in, one out" nature of both types of shader is an inherent limitation that is simplifying yet at times frustrating. A vertex shader has access to the attributes of the current vertex only. It has no knowledge of surface continuity and cannot access other vertex array elements. Similarly, the fragment shader receives and can write to only the properties of the current fragment and cannot change the screen-space position of that fragment. It cannot access neighboring fragments or the source vertices of the triangle that contains the fragment. The sole deviation from this standard is that in many shading systems, the fragment shader can generate one or zero fragments. In other words, the fragment shader can choose to "kill" the current fragment, leaving a hole in the surface. This is useful for creating intra triangle cutouts to the surface.

Looking at the pipeline depicted in Figure 7.5 in reverse, from a single-shaded fragment backwards gives an understanding of the overall pipeline as a function. If viewed in reverse (bottom to top), Figure 7.5 can demonstrate this. Starting from the end, the final, shaded fragment was computed in the fragment shader based on input values that are interpolated to the fragment's position within the triangle that contains it. This containing triangle is based upon three transformed and processed vertices that were each individually output from the vertex shader. These vertices were provided, along with the triangle connectivity, as a part of the geometry object being drawn. Thus, the entire shading pipeline is, in a sense, one long function.

7.6.2 Shader Input and Output Values

Both vertex and fragment shaders receive their inputs in roughly the same types, the most common being floating-point scalars (float in GLSL); vectors (`vec2`, `vec3`, and `vec4` in GLSL); matrices (`mat2`, `mat3`, `mat4`, etc. in GLSL); and arrays of each of these types of values. Colors are an extremely common type passed in to both forms of shaders and are generally represented in the shaders as floating-point 4-vectors, just as discussed in the introductory material in this chapter (although they are accessed in the shader as `v.r`, `v.g`, etc., instead of `v.x`, `v.y`, etc.). Integers and associated vectors and arrays are often supported as well.

One additional type of input to a shader is a texture sampler, which represents image-based lookup within the shader. This is an extremely powerful

shader input and will garner its own section later in this chapter and in the chapters to come. While some modern graphics systems and APIs allow samplers as inputs to both vertex and fragment shaders, this is not universal, and for the purposes of this book, we will discuss them as inputs to fragment shaders, where they are universally supported.

7.6.3 Shader Operations and Language Constructs

The set of shader operations in modern shading languages is generally the same in both vertex and fragment shaders. The operations and functions are too broad to list here, but include the most common infix operations (addition, subtraction, multiplication, division, negation) for scalar, vector, and matrix types and the sensible mixing thereof. A wide range of standard mathematical functions are also available, such as dot and cross products, vector normalization, trigonometric functions, etc.

Functions, procedures, conditionals, and loops are also provided in the high-level shading languages. However, since shaders are in essence SIMD (single instruction multiple data) systems, looping and branching can be expensive, especially on older hardware. However, the overall shading languages are exceedingly powerful.

7.7 Vertex Shaders

7.7.1 Vertex Shader Inputs

Vertex and fragment shaders do have slightly different sources of input, owing to their different locations in the rendering pipeline. Vertex shaders receive three basic sources of input: per-vertex attributes, per-object uniforms, and global constants. The first two can be thought of as properties of the geometry object being rendered, while the lattermost are properties and limits of the rendering hardware.

The per-vertex attributes are the elements of the object's vertex structure described above and will likely differ from vertex to vertex. Some per-vertex attributes are standard and are accessed via standard variables in the vertex shader. These are generally the attributes that carry over from the original fixed-function pipeline: position, surface normal, surface color, and texture coordinates. Others are application-specific and are custom to the shader; the high-level shading languages support this. We will focus on the standard attributes in this book, specifically, those in GLSL.

Note that in moving to a completely shader-based pipeline, OpenGL ES 2.0's GLSL-E shading language has far fewer standard, predefined vertex shader attributes and vertex/fragment uniforms than are available in the otherwise similar desktop OpenGL GLSL shading language. For example, since there is no concept of a model view matrix in OpenGL ES 2.0, there is no corresponding standard uniform. Instead, applications must pass any needed matrices via custom uniforms. Desktop OpenGL's GLSL, on the other hand, makes the model view matrix and many others available to the shading language via standard uniforms such as `gl_ModelViewProjectionMatrix`. We will make use of this feature of desktop GLSL in our examples.

In DirectX, Microsoft merges these approaches to some degree. While HLSL does not define fixed-function-related uniforms in the shading language itself, an additional "effects" system that D3D layers on top of the basic shaders allows for named uniforms to be linked to "semantics." These semantics make it possible for a general engine to automatically map the model view and projection matrices (among others) to be desired uniform declarations in the shader without having to explicitly query the named uniform in each shader. These are known collectively as "Standard Annotations and Semantics."

The per-object uniforms can be thought of as global variables and are the same value (or "uniform") across the entire object being drawn. As with attributes, some uniforms are standard and are automatically supplied by the system to every shader; common examples include the model view and projection matrices. Other uniforms are application-specific and are custom to the shader. These must be explicitly set in the rendering API by the application. Once again, we will focus on the system-provided attributes available in GLSL.

The constants are provided by the rendering API and represent hardware limits that may be of use to shaders attempting to deal with running on different platforms. Constants are just that — constant over all rendered objects.

7.7.2 Vertex Shader Outputs

One required vertex shader output value is the homogeneous (postprojection transform) vertex position. It must be written by all vertex shaders. The projected positions are required in order to generate screen-space triangles from which fragment samples can be generated.

Vertex shaders provide their other output values by writing to so-called "varying" variables. Standard (or built-in) varying values differ by API and shading language. Additional, custom varying values may be declared by a shader as well, although platforms may differ in the limited number of custom varying parameters that can be declared by a shader.

7.7.3 Basic Vertex Shaders

The simplest vertex shader simply transforms the incoming model-space vertex by the model view and projection matrix, and places the result in the required output register, as follows:

```
// GLSL
void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

This shader uses nothing but built-in vertex attributes, uniforms, and varying variables, and thus requires no declarations at all. It transforms a floating-point 4-vector (`vec4`) by a floating-point 4×4 matrix (`mat4`) and assigns the result to a 4-vector. However, this simple vertex shader provides no additional information about the surface — no normals, colors, or additional attributes. In general, we will use more complex vertex shaders.

7.7.4 Linking Vertex and Fragment Shaders

As described above, the triangle assembly stage takes sets of three processed vertices and generates triangles in screen space. Fragments on the surface of these triangles are generated, and the fragment shader is invoked upon each of these fragments. The connection between vertices and fragments is basically unbounded. Three vertices generate a triangle, but that triangle may generate many fragments (as will be discussed in Chapter 9). Or, the triangle may generate no fragments at all (e.g., if the triangle is outside of the view rectangle).

In defining the output values and types in its varying parameters, the vertex shader also provides one-half of the interface between itself and the fragment shader. In fact, vertex and fragment shaders can be written independently and need not map one-to-one with each other. As long as the varying values required by a fragment shader are all supplied by a given vertex shader (even if some of the vertex shader's varyings are unused in the fragment shader), those two shaders may be "linked" at runtime and used together. This ability to reuse a vertex or fragment shader with more than one of the other type of shader cuts down on the number of shaders that needs to be written, avoiding a combinatorial explosion.

Real applications like large-scale 3D games often spend a lot of development time having to manage the many different shaders and shading paths that exist in a complex rendering engine. Some applications use very large shaders that include all of the possible cases, branching between the various

cases using conditionals in the shader code. This can lead to large, complex shaders with a lot of conditionals whose results will differ only at the per-object level, a potentially wasteful option. Other applications generate shader source code in the application itself, as needed, compiling their shaders at runtime. This can be problematic as well, as the shader compilation takes significant CPU cycles and can stall the application visibly. Finally, some applications use a hybrid approach, generating the required shaders offline and keeping them in a lookup table, loading the required shader based on the object being rendered.

7.8 Fragment Shaders

7.8.1 Fragment Shader Inputs

Unlike vertex shaders, which are invoked on application-supplied vertices, fragment shaders are invoked on dynamically generated fragments. Thus, there is no concept of per-fragment attributes being passed into the fragment shader by the application. Varying values passed on from the vertex shader are the only unique per-fragment values.

Shader-custom varying values written by a vertex shader are simply interpolated and provided to the linked fragment shader. They must be declared in the fragment shader using the same name and type as they were declared in the vertex shader, so they can be linked together. Some of the built-in varying values written by a shader are provided in a similarly direct manner. However, others are provided in a somewhat different manner as is appropriate to the primitive and value. For example, in GLSL, the linked built-in varying values for vertex shader position output (which is specified in homogeneous coordinates) and the fragment shader's built-in fragment coordinate (which is in a window-relative coordinate) are in different spaces. Also, while the vertex shader includes predefined output varying variables for both front and back surface colors, the fragment shader is only given one of this set of colors, depending on whether the current fragment being shaded represents the front or back side of the surface.

Fragment shaders support constants and uniforms. A set of fragment shader–relevant constants may be provided by the implementation. In addition, fragment shaders can access uniform values in the same way they are accessed in vertex shaders. Fragment shaders also support an extremely powerful type of uniform value: texture image samplers (as mentioned above, some implementations support texture samplers in vertex shaders as well, but these are not as ubiquitous). These types of uniforms are so useful that we will dedicate entire sections to them in several of the rendering chapters.

7.8.2 Fragment Shader Outputs

The basic goal of the fragment shader is to compute the color of the current fragment. The entire pipeline, in essence, comes down to this single output value per fragment. The fragment shader cannot change the other values of the fragment, such as the position of the fragment, which remains locked in screen space. However, some shading systems do allow for a fragment to cancel itself, causing that fragment to go no further in the rendering pipeline. This is useful for "cutout" effects and performance optimizations.

Each shading language defines a built-in variable into which the final color must be written; in GLSL, this variable is `gl_FragColor`. An extremely basic shader that takes an application-set per-object color and applies it to the entire surface is shown below.

```
// GLSL
uniform vec4 objectColor;

void main()
{
   gl_FragColor = objectColor;
}
```

The fragment shader above is compatible with the simple vertex shader above—the two could be linked and used together.

Note that in the latest shading systems, a shader may output more than one color or value per fragment. This functionality is known as *multiple render targets* (MRTs) and will not be discussed in this text, as it does not directly affect the basic pipeline or mathematics of the system. However, the technique is extremely powerful and allows for many high-end rendering effects to be done efficiently. For details and examples of the use of MRTs, see Gray [48].

7.8.3 Compiling, Linking, and Using Shaders

Programmable shaders are analogous to many other computer programs. They are written in a high-level language (GLSL, in our case), built from multiple source files or sections (a vertex shader and a fragment shader), compiled into "machine language" (the GPU's microcode), and linked (the vertex shader together with the fragment shader). The resulting program then can be used.

This implies several stages. The first stage, compilation, can be done at runtime in the application, or may be done as an offline process. The availability of runtime compilation is dependent upon the platform. OpenGL

drivers include a GLSL compiler. Direct3D ships a runtime compiler as an independent library. OpenGL ES does not require that a platform provide a runtime compiler. However, we will assume the availability of a runtime compiler in our Iv code examples. In either case, the source vertex and fragment shaders must be compiled into compiled shader objects. If there are syntax errors in the source files, the compilation will fail.

A pair of compiled shaders (a vertex shader and a fragment shader) must then be linked into an overall shader or program. Most platforms support performing this step at runtime. Linking can fail if the vertex shader does not declare all of the varying parameters that the fragment shader requires.

For details of how OpenGL and Direct3D implement shader compilation and linking, see the source code for Iv. Depending on the rendering API, some or all of these steps may be grouped into fewer function calls. In order to compile and link source shaders into a program in Iv, the steps are shown below. Iv supports loading and compiling shaders from text file or from string. The latter case is useful for simple shaders, as they can be simply compiled into the application itself as a static string, per the following code:

```
// Shader compilation code
IvShaderProgram* LoadProgram(IvResourceManager& manager)
{
   IvVertexShader* vertexShader
      = manager.CreateVertexShaderFromFile("vert.txt");
   IvFragmentShader* fragmentShader
      = manager.CreateFragmentShaderFromFile("frag.txt");

   IvShaderProgram* program
      = manager.CreateShaderProgram(vertexShader, fragmentShader);

   return program;
}
```

The resulting program object then must be set as the current shading program before an object can be rendered using it. In Iv, the code to set the current shading program is as follows. Other APIs use similar function calls, as follows:

```
IvResourceManager& manager; IvRenderer& renderer;
IvShaderProgram* program;
// ...

   // Shader apply code
   renderer.SetShaderProgram( program );
```

7.8.4 Setting Uniform Values

As mentioned previously, uniform shader parameters form the most immediate application-to-shader communication. These values provide the "global" variables required inside of a shader and can be set on a per-object basis. Since they cannot be set during the course of a draw call, there is no way to change uniforms at a finer grain than the per-object level. Only per-vertex attributes (in the vertex shader) and varyings (in the fragment shader) will differ at that fine-grained level.

The first step in being able to set a uniform value for a shader is to query the uniform value by name from the application. Rendering APIs that support high-level shading languages also support some method of mapping string names for uniforms into the uniforms themselves. The exact method differs from API to API. However, querying by string can be expensive and should not be done every time an application needs to access a uniform in a shader. As a result, the rendering APIs can, given a string name and a shading program object, return a "handle" or pointer to an object that represents the uniform. While the initial lookup still requires a string match, the returned handle allows the uniform to be changed later without a string lookup each time. In `Iv`, the query function is as follows:

```
IvShaderProgram* program;
// ...

   IvUniform* uniform = program->GetUniform("myShaderUniformName");
```

The handle variable `uniform` now represents that uniform *in that shader* from this point onward. Note that uniforms are in the scope of a given shading program. Thus, if you need to set a uniform in multiple shading programs, you will need to query the handles and set the values independently for each shading program, even if the uniform has the same name in all of the programs. Although the application will generally know the type of the uniform already (since the application developer likely wrote the shader code), rendering APIs make it possible to retrieve the type (float; integer; Boolean; 2-, 3-, and 4-vectors of each; and float matrices) and array count (one or more of each type) for a uniform. Finally, the rendering API will include functions to set (and perhaps get) the values of each uniform. `Iv` code that demonstrates querying the type and count of a uniform as well as setting the value is as follows. The code below queries a handle for a uniform that is known to be a two-element array of 4D vectors, perhaps representing a pair of basis vectors.

```
IvUniform* uniform;
// ...
```

```
IvUniformType uniformType = uniform->GetType();
unsigned int uniformCount = uniform->GetCount();

// We're expecting an array of two float vector-4's
if ((uniformType == kFloat4Uniform) &&
    (uniformCount == 2))
{
   // Set the vectors to the Z and X axes
   uniform->SetValue(IvVector4(0, 0, 1, 0), 0);
   uniform->SetValue(IvVector4(1, 0, 0, 0), 1);
}
```

These interfaces make it possible to pass a wide range of data items down from the application code to a shader. We will use uniforms extensively in Chapter 8 as we discuss lighting. Uniforms will form the basis of how we pass information regarding the number, type, and configuration of lights and surfaces to the shaders that will actually compute the lit colors.

7.9 Basic Coloring Methods

The following sections describe a range of simple methods to assign colors to surface geometry. Note that the cases described below are designed to best explain how to pass the desired colors to the fragment shader and are overly simplified. These basic methods can be (and will be in later sections and chapters) used to pass other noncolor values into the fragment shader for more complex shading. However, this initial discussion will focus simply on passing different forms of color values to the fragment shader, which will in turn simply write the color value being discussed directly as its output.

The simplest and generally highest-performing methods of coloring geometry are to use constant colors. Constant colors involve "passing through" colors that were assigned to the geometry prior to rendering. These colors may have been generated by having an artist assign colors to every surface during content creation time. Alternatively, an offline process may have been used to generate static colors for all geometry. With these static colors assigned, there is relatively little that must be done to select the correct color for a given fragment. Constant colors mean that for a given piece of geometry, the color at a fixed point on the surface will never change. No environmental information like dynamic lighting will be factored into the final color.

The following examples will show simple cases of constant color. These will serve as building blocks for later dynamic coloring methods, such as lighting.

7.9.1 Per-Object Colors

The simplest form of useful coloring is to assign a single color per object. Constant coloring of an entire object is of very limited use, since the entire object will appear to be flat, with no color variation. At best, only the filled outline of the object will be visible against the backdrop. As a result, except in some special cases, per-object color is rarely used as the final shading function for an object.

Per-object color requires no special work in the vertex shader (other than basic projection). The vertex/fragment shader pair below implements per-object colors. The application need only specify the desired color by setting the color into the named uniform `objectColor`. The `objectColor` uniform must be declared in the fragment shader and the application must set its value for the current object prior to rendering the object; it is not a built-in uniform.

```
// GLSL
void main() // vertex shader
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
// GLSL
uniform vec4 objectColor;

void main() // fragment shader
{
   gl_FragColor = objectColor;
}
```

7.9.2 Per-Vertex Colors

Many of the surfaces approximated by tessellated objects are smooth, meaning that the goal of coloring these surfaces is to emphasize the smoothness of the original surface, not the artifacts of its approximation with flat triangles. This fact makes flat shading a very poor choice for many tessellated objects. A shading method that can generate the appearance of a smooth surface is needed. Per-vertex coloring, along with a method called *Gouraud shading* (after its inventor, Henri Gouraud) does this. Gouraud shading is based on the existence of some form of per-vertex colors, assigning a color to any point on a triangle by linearly interpolating the three vertex colors over the surface of the triangle. As with the other shading methods we have discussed, Gouraud shading is independent of the source of these per-vertex colors; the

vertex colors may be assigned explicitly by the application, or generated on-the-fly via per-vertex lighting or other vertex shader. This linear interpolation is both simple and smooth and can be expressed as a mapping of barycentric coordinates (s, t) as follows:

$$Color(O, T, (s, t)) = sC_{V1} + tC_{V2} + (1 - s - t)C_{V3}$$

Examining the terms of the equation, it can be seen that Gouraud shading is simply an affine transformation from barycentric coordinates (as homogeneous points) in the triangle to RGB color space.

An important feature of per-vertex smooth colors is that color discontinuities can be avoided at triangle edges, making the piecewise-flat tessellated surface appear smooth. Internal to each triangle, the colors are interpolated smoothly. At triangle edges, color discontinuities can be avoided by ensuring that the two vertices defining a shared edge in one triangle have the same color as the matching pair of vertices in the other triangle. It can be easily shown that at a shared edge between two triangles, the color of the third vertex in each triangle (the vertices that are not an endpoint of the shared edge) does not factor into the color along that shared edge. As a result, there will be no color discontinuities across triangle boundaries, as long as the shared vertices between any pair of triangles are the same in both triangles. In fact, with fully shared, indexed geometry, this happens automatically (since colocated vertices are shared via indexing). Figure 7.6 allows a comparison of geometry drawn with per-face colors and with per-vertex colors.

Per-vertex colors are generated in the vertex shader, either through computation, direct use of per-vertex attributes, or a combination of both. In the fragment shader, the built-in vertex color-varying value (which has been interpolated to the correct value for the fragment using Gouraud interpolation) is used directly.

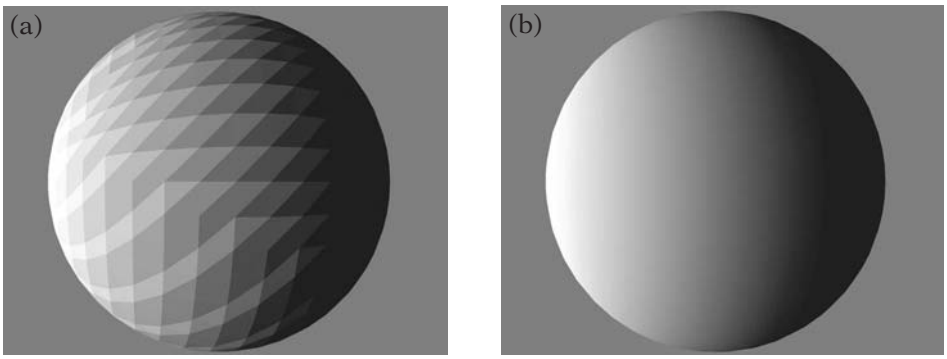


FIGURE 7.6 (a) Flat (per-face) and (b) Gouraud (per-vertex) shading.

```
// GLSL
void main() // vertex shader
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_FrontColor = gl_BackColor = gl_Color;
}

// GLSL
void main() // fragment shader
{
   gl_FragColor = gl_Color;
}
```

7.9.3 Per-Triangle Colors

Rounding out the "primitive-level" coloring methods is per-triangle coloring. This method simply assigns a color to each triangle. This is also known as *faceted*, or *flat*, shading, because the resulting geometry appears planar on a per-triangle basis. Technically, this requires adding a color attribute for each triangle. However, explicit per-triangle attributes are not supported in most current rendering systems. As a result, in order to support per-triangle colors, rendering APIs tend to allow for a mode in which the color value computed for one of a triangle's vertices is used as the varying value for the entire triangle, with no interpolation.

There are two common ways of specifying flat shading in programmable shading APIs. A shader-external render-state setting may be used to place the rendering pipeline in flat-shaded mode. This is the method used by `Iv`, enabled via the `IvRenderer` function `SetShadeMode`. The single argument to this function sets the shading mode: `kFlatShadeMode` sets flat shading and `kSmoothShadeMode` sets Gouraud shading. Having placed the system into flat-shaded mode, the triangle assembly stage will automatically duplicate the vertex color-varying value(s) from one of the triangle's vertices to the other two, causing all fragments for that triangle to receive the same color(s). The other method of specifying per-triangle constant colors is built into the shading language itself, whereby a varying value is declared in the shader with a "flat"-type modifier. Varying values declared as "flat" will not be interpolated before being passed down to the fragment shader.

7.9.4 Sharp Edges and Vertex Colors

Many objects that we render will contain a mixture of smooth surfaces and sharp edges. One need only look at the outlines of a modern automobile to

see this mixture of sloping surfaces (a rounded fender) and hard creases (the sharp edge of a wheelwell). Such an object cannot be drawn using per-triangle colors, as per-triangle colors will correctly represent the sharp edges, but will not be able to represent the smooth sections. In these kinds of objects, *some* sharp geometric edges in the tessellation really do represent the original surface accurately, while other sharp edges are designed to be interpolated across to approximate a smooth section of surface.

In addition, the edge between two triangles may mark the boundary between two different colors on the surface of the object, such as an object with stripes painted upon it. In this context, a "sharp" edge is not necessarily a geometric property. It is nothing more than an edge that is shared by two adjacent triangles where the triangle colors on either side of the edge are different. This produces a visible, sharp line between the two triangles where the color changes.

In these situations, we must use per-vertex interpolated colors. However, interpolating smoothly across all triangle boundaries is not the desired behavior with a smooth/sharp object. The vertices along a sharp edge need to have different colors in the two triangles abutting the edge. In general, when Gouraud shading is used, these situations require coincident vertices to be duplicated, so that the two coincident copies of the vertex can have different colors. Figure 7.7 provides an example of a cube drawn with entirely shared vertices and with duplicated vertices to allow per-vertex, per-face colors. Note that the cube is not flat-shaded in either case — there are still color gradients across each face. The example with duplicated vertices and sharp shading edges looks more like a cube.

7.9.5 More about Basic Shading

For far more details on the rendering of flat- versus smooth- (or Gouraud) shaded triangles, see Chapter 9. Both flat and Gouraud shading are used to

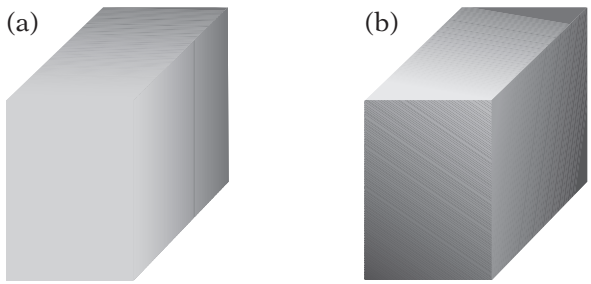


Figure 7.7 Sharp vertex discontinuities: (a) shared vertices lead to smooth-shaded edges, and (b) duplicated vertices allow the creation of sharp-shaded edges.

interpolate colors generated by dynamic lighting. For a detailed discussion of dynamic lighting, see Chapter 8.

7.9.6 Limitations of Basic Shading Methods

Real-world surfaces often have detail at many scales. The shading/coloring methods described so far require that the fragment shader compute a final color based solely on sources assigned at tessellation-level features, either per-triangle or pervertex. While this works well for surfaces whose colors change at geometric boundaries, many surfaces do not fit this restriction very well, making flat shading and Gouraud shading ineffective at best. While programmable shaders can be used to compute very complex coloring functions that change at a much higher frequency than per-vertex or per-triangle methods, doing so based only on these gross-scale inputs can be difficult and inefficient.

For example, imagine a flat sheet of paper with text written on it. The flat, rectangular sheet of paper itself can be represented by as few as two triangles. However, in order to use Gouraud shading (or even more complex fragment shading based on Gouraud-interpolated sources) to represent the text, the piece of paper would have to be subdivided into triangles at the edges of every character written on it. None of these boundaries represents geometric features, but rather are needed only to allow the color to change from white (the paper's color) to black (the color of the ink). Each character could easily require hundreds of vertices to represent the fine stroke details. This could lead to a simple, flat piece of paper requiring tens of thousands of vertices. Clearly, we require a shading method that is capable of representing detail at a finer scale than the level of tessellation.

7.10 Texture Mapping

7.10.1 Introduction

One method of adding detail to a rendered image without increasing geometric complexity is called *texture mapping*, or more specifically *image-based texture mapping*. The physical analogy for texture mapping is to imagine wrapping a flat, paper photograph onto the surface of a geometric object. While the overall shape of the object remains unchanged, the overall surface detail is increased greatly by the image that has been wrapped around it. From some distance away, it can be difficult to even distinguish what pieces of visual

detail are the shape of the object and which are simply features of the image applied to the surface.

A real-world physical analogy to this is theatrical set construction. Often, details in the set will be painted on planar pieces of canvas, stretched over a wooden frame (i.e.,"flats"), rather than built out of actual, 3D wood, brick, or the like. With the right lighting and positioning, these painted flats can appear as very convincing replicas of their real, 3D counterparts. This is the exact idea behind texturing—using a 2D, detailed image placed upon a simple 3D geometry to create the illusion of a complex, detailed, fully 3D object.

An example of a good use of texturing is a rendering of a stucco wall; such a wall appears flat from any significant distance, but a closer look shows that it consists of many small bumps and sharp cracks. While each of these bumps could be modeled with geometry, this is likely to be expensive and unlikely to be necessary when the object is viewed from a distance. In a 3D computer graphics scene, such a stucco wall will be most frequently represented by a flat plane of triangles, covered with a detailed image of the bumpy features of lit stucco.

The fact that texture mapping can reduce the problem of generating and rendering complex 3D objects into the problem of generating and rendering simpler 3D objects covered with 2D paintings or photographs has made texture mapping very popular in real-time 3D. This, in turn, has led to the method being implemented in display hardware, making it even less expensive computationally. The following sections will introduce and detail some of the concepts behind texture mapping, some mathematical bases underlying them, and basics of how texture mapping can be used in 3D applications.

7.10.2 Shading via Image Lookup

The real power of texturing lies in the fact that it uses a dense plane of samples (an image) as its means of generating color. In a sense, texturing can be thought of as a powerful, general function that maps 2-vectors (the texture coordinates) into a vector-valued output (most frequently an RGBA color). To the shader it is basically irrelevant how the function is computed. Rather than directly interpolating colors that are stored in the vertices, the interpolated per-vertex texture coordinate values serve only to describe how an image is mapped to the triangle. While the mapping from the surface into the space of the image is linear, the lookup of the image value is not. By adding this level of indirection between the per-vertex values and the final colors, texturing can create the appearance of a very complex shading function that is actually no more than a lookup into a table of samples.

The process of texturing involves defining three basic mappings:

1. To map all points on a surface (smoothly in most neighborhoods) into a 2D (or in some cases, 1D or 3D) domain.
2. To map points in this (possibly unbounded) domain into a unit square (or unit interval, cube, etc.).
3. To map points in this unit square to color values.

The first stage will be done using a modification of the method we used for colors with Gouraud shading, an affine mapping. The second stage will involve methods such as min, max, and modulus. The final stage is the most unique to texturing and involves mapping points in the unit square into an image. We will begin our discussion with a definition of texture images.

7.10.3 Texture Images

The most common form of texture images (or *textures*, as they are generally known) are 2D, rectangular arrays of color values. Every texture has a width (the number of color samples in the horizontal direction) and a height (the number of samples in the vertical direction). Textures are similar to almost any other digital image, including the screen, which is also a 2D array of colors. Just as the screen has pixels (for picture elements), textures have *texels* (texture elements). While some graphics systems allow 1D textures (linear arrays of texels) and even 3D textures (cubes or rectangular parallelepipeds of texels), by far the most common and most useful are 2D, image-based textures. Our discussion of texturing will focus entirely on 2D textures.

We can refer to the position of a given texel via a 2D value (x, y) in texel units. (Note that these coordinates are (column, row), the reverse of how we generally refer to matrix elements in our row-major matrix organization.) Figure 7.8 shows an example of a common mapping of texel coordinates into a texture. Note that while the left to right increasing mapping of x is universal in graphics systems, the mapping of y is not; top to bottom is used in Direct3D, and bottom to top is used in OpenGL.

As with most other features, while there are minor differences between the rendering APIs regarding how to specify texture images, all of the APIs require the same basic information:

- The per-texel color storage format of the incoming texture data.
- The width and height of the image in texels.
- An array of width $\times$ height color values for the image data.

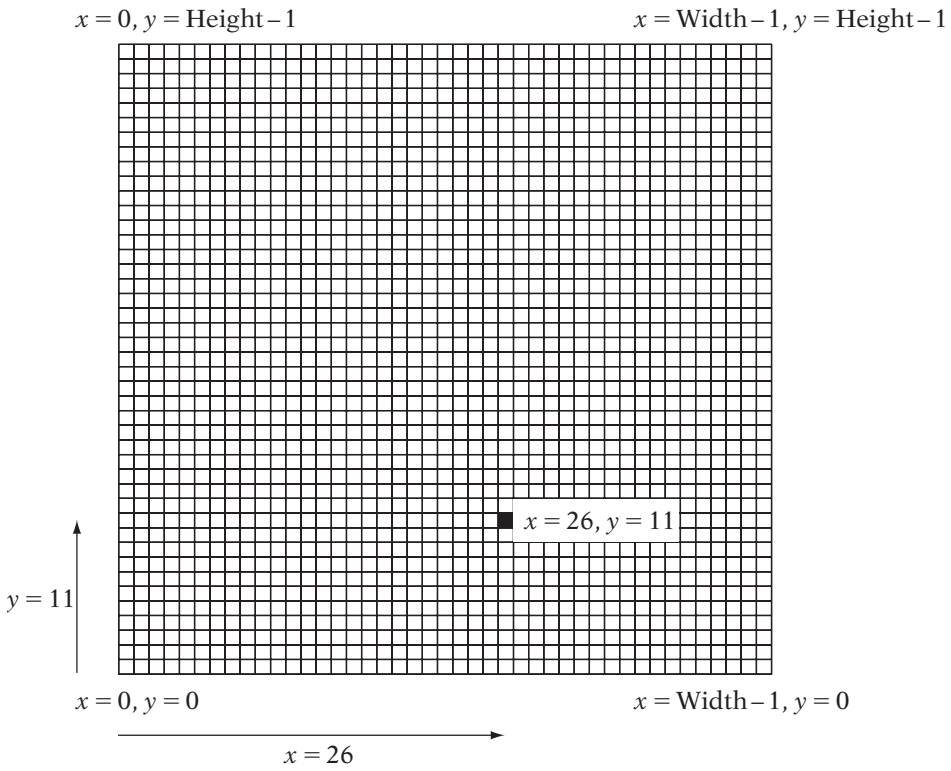


FIGURE 7.8 Texel-space coordinates in an image.

Put together, these define the image data and their basic interpretation in the same way that an array of vertices, the vertex format information, and the vertex count define vertex geometry to the rendering pipeline. As with vertex arrays, the array of texel data can be quite sizable. In fact, texture image data are one of the single-largest consumers of memory-related resources.

Rendering APIs generally include the notion of an opaque handle to a device-resident copy of a texture. For peak performance on most systems, texture image data need to reside in GPU device memory. Thus, in a process analogous to vertex buffer objects, rendering APIs include the ability to transfer a texture's image data to the device memory once. The opaque handle then can be used to reference the texture in later drawing calls, using the already-resident copy of the texture image data in GPU memory. In `Iv`, we use an object to wrap all of this state: `IvTexture`, which represents the texture image itself and the texture sampler state. Like most other resources (e.g., vertex and index buffers), `IvTexture` objects are created via the `IvResourceManager` object, as follows:

```
IvResourceManager* manager;

// ...

{
   const unsigned int width = 256;
   const unsigned int height = 512;
   IvTexture* texture = manager->CreateTexture(kRGBA32TexFmt,
                                               width, height);

   // ...
```

The preceeding code creates a texture object with a 32-bit-per-texel RGBA texture image that has a width of 256 texels and a height of 512 texels. Note that while this function allocates the texture, it does not fill it with image data. In order to fill the texture with texel data, we must "lock" the texture and write the data to the allocated memory in a manner analogous to the way we initialized vertex arrays. The code to fill an RGBA texture with bright red texels is as follows:

```
IvTexture* texture;

// ...

{
   const unsigned int width = texture->GetWidth();
   const unsigned int height = texture->GetHeight();

   IvTexColorRGBA* texels = texture->BeginLoadData();

   for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
         IvTexColorRGBA& texel = texels[x + y * width];
         texel.r = 255;
         texel.g = 0;
         texel.b = 0;
         texel.a = 255;
      }
   }

// ...
   texture->EndLoadData();
```

7.10.4 Texture Samplers

Textures appear in the shading language in the form of a texture sampler object. Texture samplers are passed to a fragment shader as a uniform value (which is a handle that represents the sampler). The same sampler can be used multiple times in the same shader, passing different texture coordinates to each lookup. So, a shader can sample a texture at multiple locations when computing a single fragment. This is an extremely powerful technique that is used in many advanced shaders. From within a shader, a texture sampler is a sort of "function object" that can be evaluated as needed, each time with unique inputs.

Texture Samplers in Application Code

At the application C or C++ level, there is considerably more to a texture sampler. A texture sampler at the API level includes at least the following information:

- The texture image data.
- Settings that control how the texture coordinates are mapped into the image.
- Settings that control how the resulting image sample is to be post-processed before returning it to the shader.

All of these settings are passed into the rendering API by the application prior to using the texture sampler in a shader. As with other shader uniforms, we must include application C or C++ code to link a value to the named uniform; in this case, the uniform value represents a texture image handle. We will cover each of these steps in the following sections.

The book's rendering API uses the `IvTexture` object to represent texture samplers and all of their related rendering state. The code examples in the following section below all describe the `IvTexture` interfaces.

7.11 Texture Coordinates

While textures can be indexed by 2D vectors of nonnegative integers on a per-texel basis (texel coordinates), textures are normally addressed in a more general, texel-independent manner. The texels in a texture are most often addressed via width- and height-independent U and V values. These

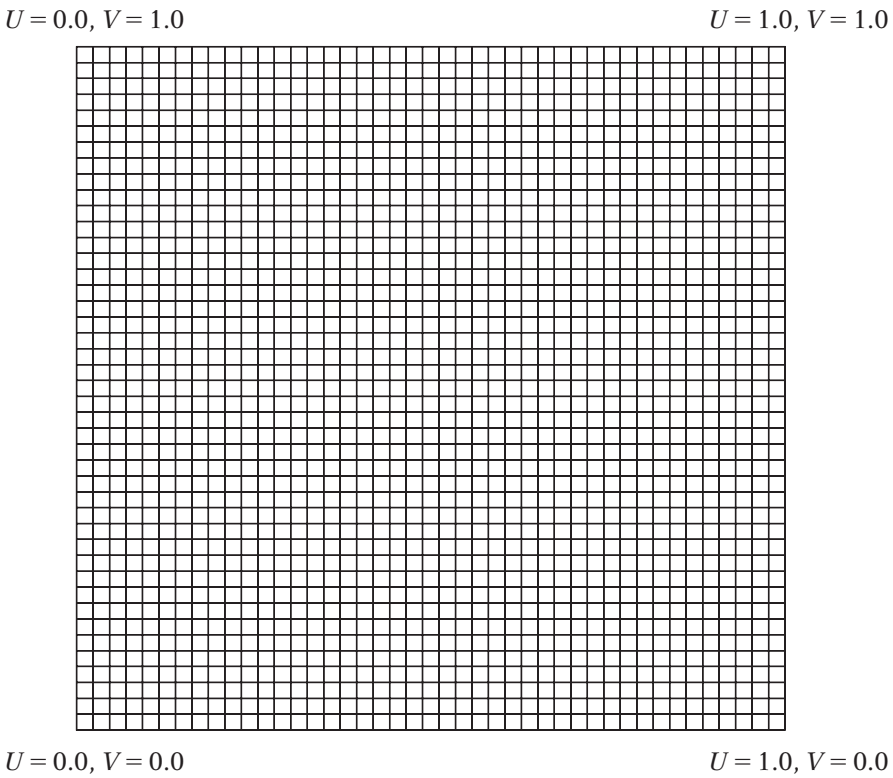


Figure 7.9 Mapping U and V coordinates into an image.

2D real-valued coordinates are mapped in the same way as texel coordinates, except for the fact that U and V are normalized, covering the entire texture with the 0-to-1 interval. Figure 7.9 depicts the common mapping of UV coordinates into a texture. These normalized UV coordinates have the advantage that they are completely independent of the height and width of the texture, meaning that the texture resolution can change without having to change the mapping values. Almost all texturing systems use these normalized UV coordinates at the application and shading language level, and as a result, they are often referred to by the generic term of *texture coordinates*, or *texture UVs*.

7.11.1 Mapping Texture Coordinates onto Objects

The texture coordinates defined at the three vertices of a triangle define an affine mapping from barycentric coordinates to UV space. Given the

barycentric coordinates of a point in a triangle, the texture coordinates may be computed as

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (u_{V1} - u_{V3}) & (u_{V2} - u_{V3}) & u_{V3} \\ (v_{V1} - v_{V3}) & (v_{V2} - v_{V3}) & v_{V3} \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

Although there is a wide range of methods used to map textures onto triangles (i.e., to assign texture coordinates to the vertices), a common goal is to avoid "distorting" the texture. In order to discuss texture distortion, we need to define the U and V basis vectors in UV space. If we think of the U and V vectors as 2-vectors rather than the "pointlike" texture coordinates themselves, then we compute the basis vectors as

$$\mathbf{e}_u = (1, 0) - (0, 0)$$
$$\mathbf{e}_v = (0, 1) - (0, 0)$$

The $\mathbf{e}_u$ vector defines the mapping of the horizontal dimension of the texture (and its length defines the size of the mapped texture in that dimension), while the $\mathbf{e}_v$ vector does the same for the vertical dimension of the texture.

If we want to avoid distorting a texture when mapping it to a surface, we must ensure that the affine mapping of a texture onto a triangle involves rigid transforms only. In other words, we must ensure that these texture-space basis vectors map to vectors in object space that are perpendicular and of equal length. We define *ObjectSpace*() as the mapping of a vector in texture space to the surface of the geometry object. In order to avoid distorting the texture on the surface, *ObjectSpace*() should obey the following guidelines:

$$ObjectSpace(\mathbf{e}_u) \cdot ObjectSpace(\mathbf{e}_v) = 0$$
$$|ObjectSpace(\mathbf{e}_u)| = |ObjectSpace(\mathbf{e}_v)|$$

In terms of an affine transformation, the first constraint ensures that the texture is not sheared on the triangle (i.e., perpendicular lines in the texture image will map to perpendicular lines in the plane of the triangle), while the second constraint ensures that the texture is scaled in a uniform manner (i.e., squares in the texture will map to squares, not rectangles, in the plane of the triangle). Figure 7.10 shows examples of texture-to-triangle mappings that do not satisfy these constraints.

Note that these constraints are by *no* means a requirement — many cases of texturing will stray from them, through either artistic desire or the simple mathematical inability to satisfy them in a given situation. However, the degree that these constraints do hold true for the texture coordinates on a

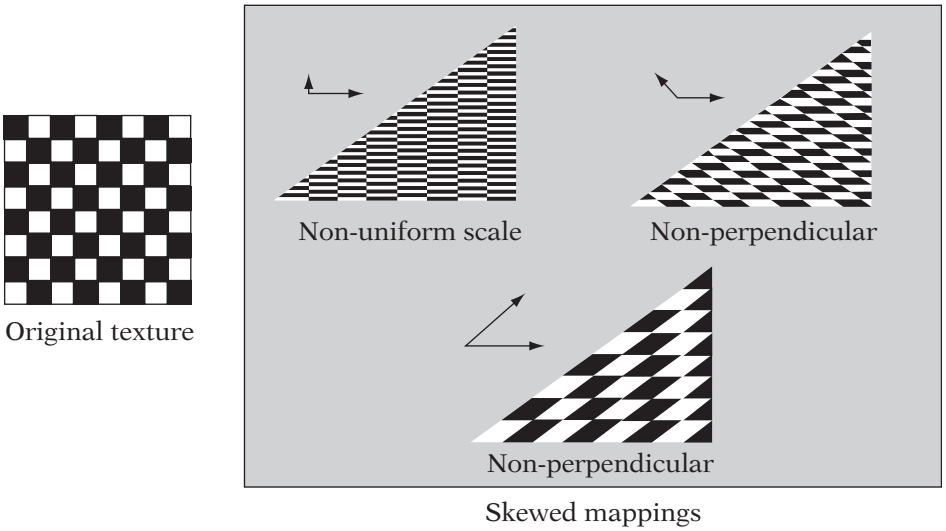


Figure 7.10 Examples of "skewed" texture coordinates.

triangle give some measure of how closely the texturing across the triangle will reflect the original planar form of the texture image.

7.11.2 Generating Texture Coordinates

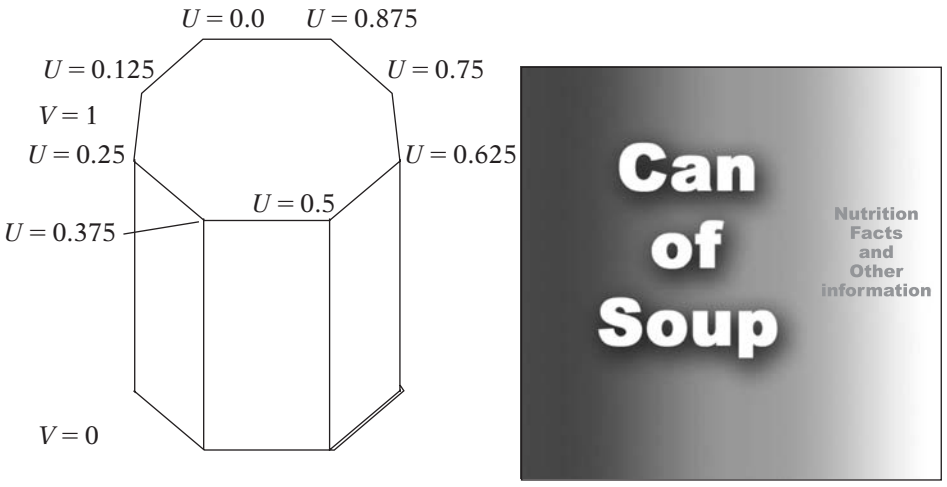
Texture coordinates are often generated for an object by some form of projection of the object-space vertex positions in $\mathbb{R}^3$ into the per-vertex texture coordinates in $\mathbb{R}^2$. All texture coordinate generation — in fact, all 2D texturing — is a type of projection. For example, imagine the cartographic problem of drawing a flat map of Earth. This problem is directly analogous to mapping a 2D texture onto a spherical object. The process cannot be done without distortion of the texture image. Any 2D texturing of a sphere is an exercise in matching a projection/"unwrapping" of the sphere onto a rectangular image (or several images) and the creation of 2D images that take this mapping into account. For example, a common, simple mapping of a texture onto a sphere is to use U and V as longitude and latitude, respectively, in the texture image. This leads to discontinuities at the poles, where more and more texels are mapped over smaller and smaller surface areas as we approach the poles.

The artist must take this into account when creating the texture image. Except for purely planar mappings (such as the wall of a building), most texturing work done by an artist is an artistic cycle between generating texture coordinates upon the object and painting textures that are distorted correctly to map in the desired way to those coordinates.

7.11.3 Texture Coordinate Discontinuities

As was the case with per-vertex colors, there are situations that require shared, collocated vertices to be duplicated in order to allow the vertices to have different texture coordinates. These situations are less common than in the case of per-vertex colors, due to the indirection that texturing allows. Pieces of geometry with smoothly mapped texture coordinates can still allow color discontinuities on a per-sample level by painting the color discontinuities into the texture. Normally, the reason for duplicating collocated vertices in order to split the texture coordinates has to do with topology.

For example, imagine applying a texture as the label for a model of a tin can. For simplicity, we shall ignore the top and bottom of the can and simply wrap the texture as one would a physical label. The issue occurs at the texture's seam. Figure 7.11 shows a tin can modeled as an eight-sided cylinder containing 16 shared vertices—8 on the top and 8 on the bottom. The mapping in the vertical direction of the can (and the label) is simple, as shown in the figure. The bottom 8 vertices set $V = 0.0$ and the top 8 vertices set $V = 1.0$. So far, there is no problem. However, problems arise in the assignment of U. Figure 7.12 shows an obvious mapping of U to both the top and bottom vertices — U starts at 0.0 and increases linearly around the can until the eighth vertex, where it is 0.875, or $1.0 - 0.125$.



Shared vertex *UV*s Texture image

Figure 7.11 Texturing a can with completely shared vertices.

Front side
(Appears to be correctly mapped)

Back side
(Incorrect, due to shared
vertices along the label "seam")

Figure 7.12 Shared vertices can cause texture coordinate problems.

The problem is between the eighth vertex and the first vertex. The first vertex was originally assigned a U value of 0.0, but at the end of our circuit around the can, we would also like to assign it a texture coordinate of 1.0, which is not possible for a single vertex. If we leave the can as is, most of it will look perfectly correct, as we see in the front view of Figure 7.12. However, looking at the back view in Figure 7.12, we can see that the face between the eighth and first vertex will contain a squashed version of almost *the entire texture, in reverse!* Clearly, this is not what we want (unless we can always hide the seam). The answer is to duplicate the first vertex, assigning the copy associated with the first face $U = 0.0$ and the copy associated with the eighth face $U = 1.0$. This is shown in Figure 7.13 and looks correct from all angles.

7.11.4 Mapping Outside the Unit Square

Source Code
Demo
TextureAddressing

So far, our discussion has been limited to texture coordinates within the unit square, $0.0 \leq u$ and $v \leq 1.0$. However, there are interesting options available if we allow texture coordinates to fall outside of this range. In order for this to work, we need to define how texture coordinates map to texels in the texture when the coordinates are less than 0.0 or greater than 1.0. These operations are per sample, not per vertex, as we shall discuss.

Front side
(Correct: unchanged from
previous mapping)

Back side
(Correct, due to doubled
vertices along the label "seam")

FIGURE 7.13 Duplicated vertices used to solve texturing issues.

The most common method of mapping unbounded texture coordinates into the texture is known as *texture wrapping*, *texture repeating*, or *texture tiling*. The wrapping of a component u of a texture coordinate is defined as

$$wrap(u) = u - \lfloor u \rfloor$$

The result of this mapping is that multiple "copies" of the texture "tile" the surface. Wrapping must be computed using the per sample, not per-vertex, method. Figure 7.14 shows a square whose vertex texture coordinates are all outside of the unit square, with a texture applied via per-sample wrapping. Clearly, this is a very different result than if we had simply applied the wrapping function to each of the vertices, which can be seen in Figure 7.15. In most cases, per-vertex wrapping produces incorrect results.

Wrapping is often used to create the effect of a tile floor, paneled walls, and many other effects where obvious repetition of a texture is required. However, in other cases wrapping is used to create a more subtle effect, where the edges of each copy of the texture are not quite as obvious. In order to make the edges of the wrapping less apparent, texture images must be created in such a way that the matching edges of the texture image are equal.

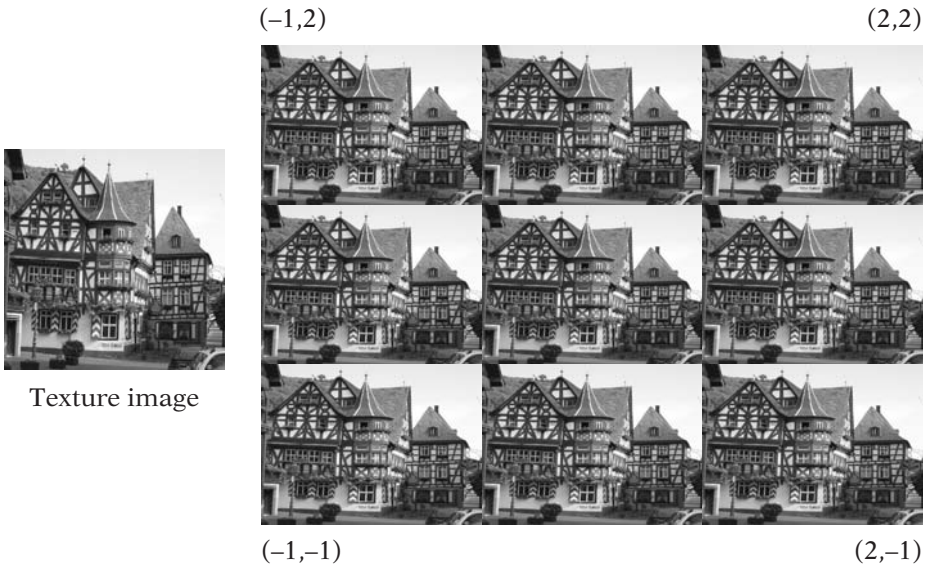


FIGURE 7.14 An example of texture wrapping.

Wrapping creates a toroidal mapping of the texture, as tiling matches the bottom edge of the texture with the top edge of the neighboring copy (and vice versa), and the left edge of the texture with the right edge of the neighboring copy (and vice versa). This is equivalent to rolling the texture into a tube (matching the top and bottom edges), and then bringing together the ends of the tube, matching the seams. Figure 7.16 shows this toroidal matching of texture edges. In order to avoid the sharp discontinuities at the texture repetition boundaries, the texture must be painted or captured in such a way that it has "toroidal topology"; that is, the neighborhood of its top edge is equal to the neighborhood of its bottom edge, and the neighborhood of its left edge must match the neighborhood of its right edge. Also, the neighborhood of the four corners must be all equal, as they come together in a point in the mapping. This can be a tricky process for complex textures, and various algorithms have been built to try to create toroidal textures automatically. However, the most common method is still to have an experienced artist create the texture by hand to be toroidal.

The other common method used to map unbounded texture coordinates is called *texture clamping*, and is defined as

$$clamp(u) = max(min(u, 1.0), 0.0)$$

Clamping has the effect of simply stretching the border texels (left, right, top, and bottom edge texels) out across the entire section of the triangle that falls

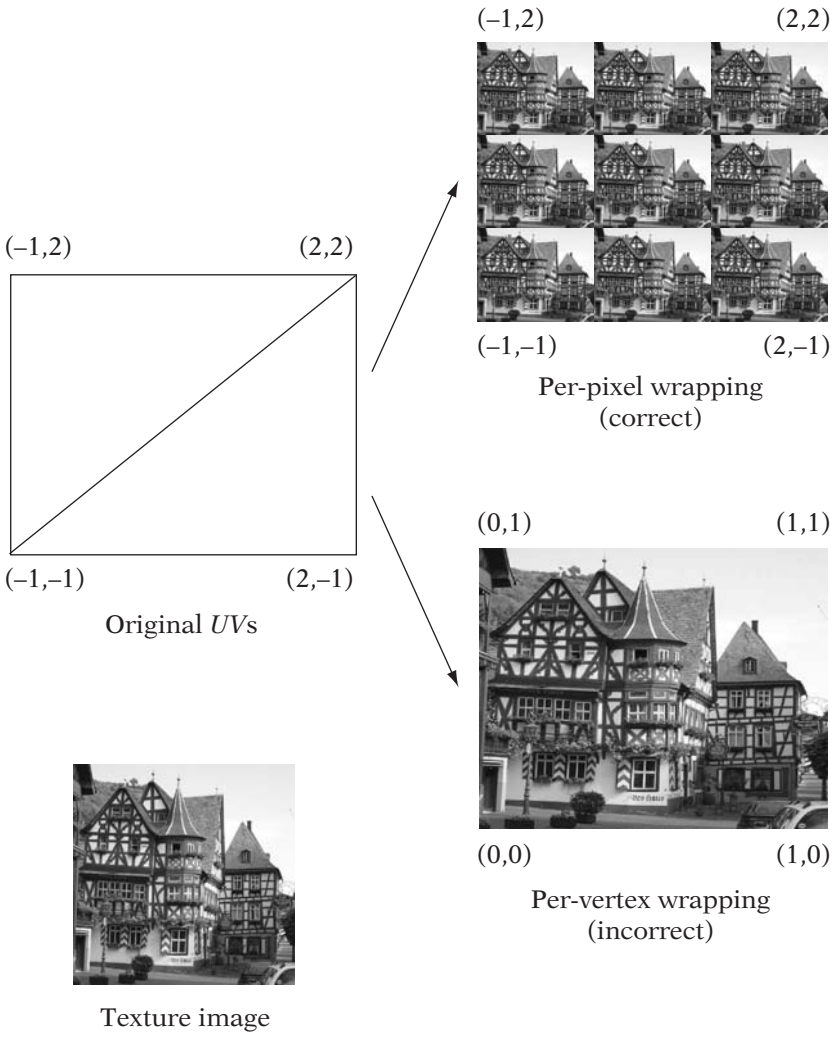


FIGURE 7.15 Computing texture wrapping.

outside of the unit square. An example of the same square we've discussed, but with texture clamping instead of wrapping, is shown in Figure 7.17. Note that clamping the vertex texture coordinates is very different from texture clamping. An example of the difference between these two operations is shown in Figure 7.18. Texture clamping must be computed per sample and has no effect on any sample that would be in the unit square. Per-vertex coordinate clamping, on the other hand, affects the entire mapping to the triangle, as seen in the lower-right corner of Figure 7.18.

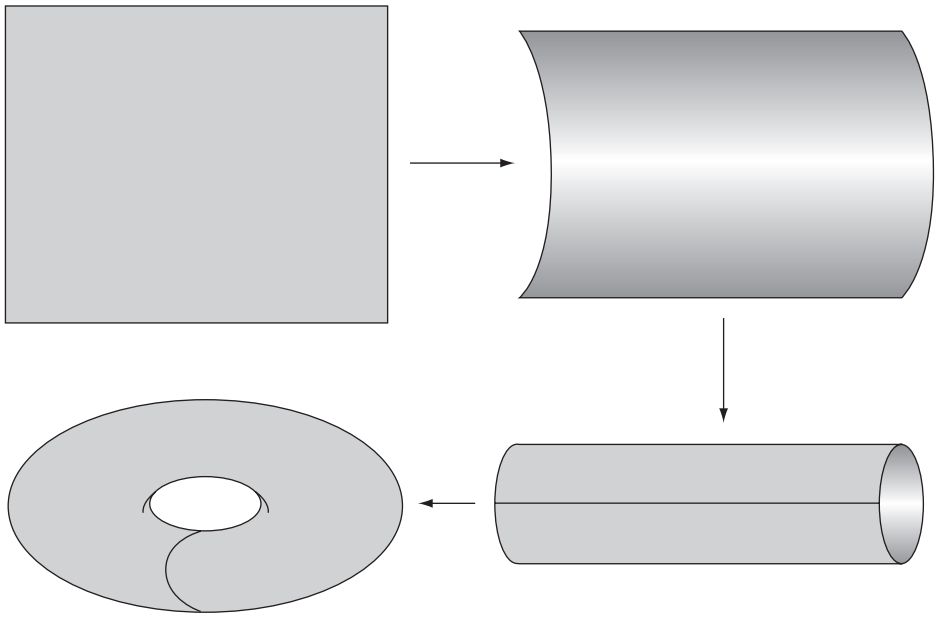


FIGURE 7.16 Toroidal matching of texture edges when wrapping.

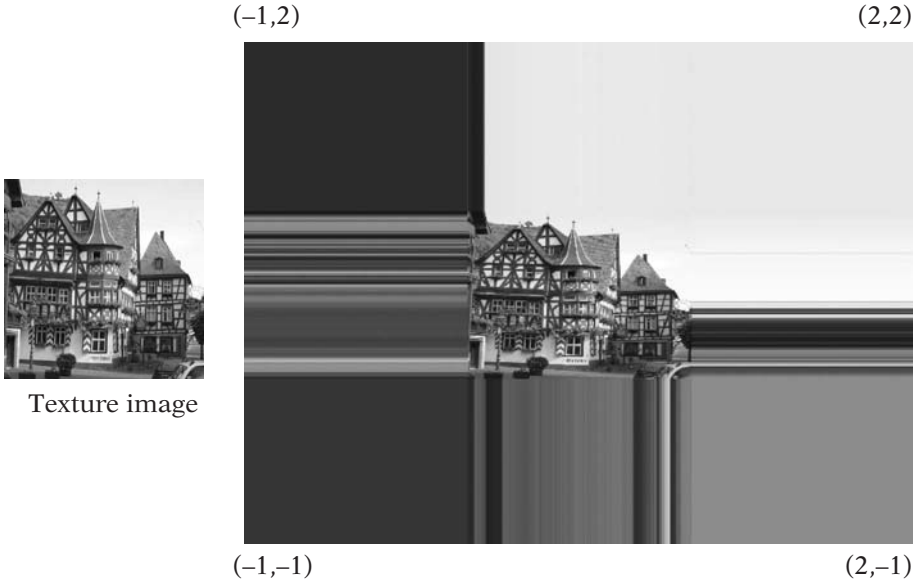


FIGURE 7.17 An example of texture clamping.

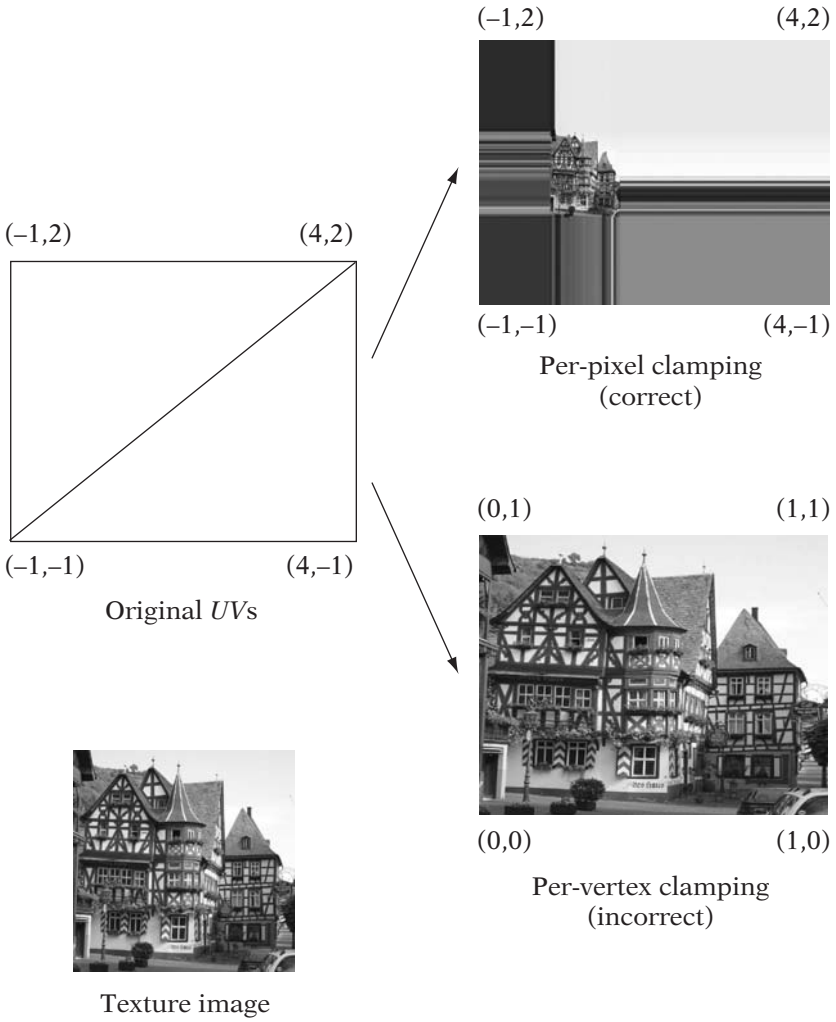


FIGURE 7.18 Computing texture clamping.

Clamping is useful when the texture image consists of a section of detail on a solid-colored background. Rather than wasting large expanses of texels and placing a small copy of the detailed section in the center of the texture, the detail can be spread over the entire texture but leaving the edges of the texture as the background color.

On many systems clamping and wrapping can be set independently for the two dimensions of the texture. For example, say we wanted to create the effect of a road: black asphalt with a thin set of lines down the center of the road. Figure 7.19 shows how this effect can be created with a very small texture

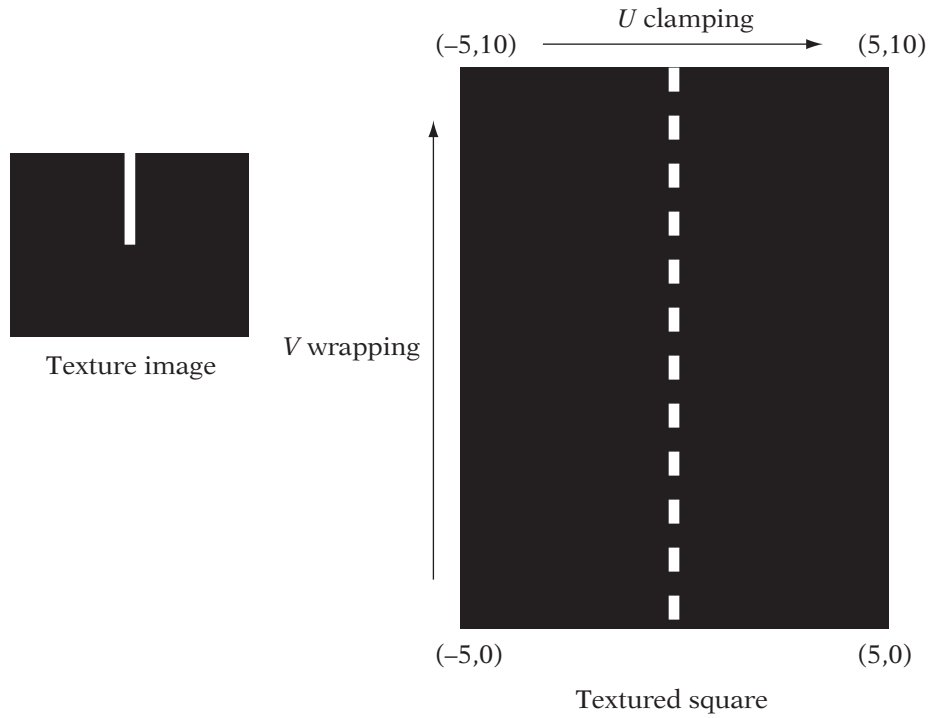


FIGURE 7.19 Mixing clamping and wrapping in a useful manner.

by clamping the *U* dimension of the texture (to allow the lines to stay in the middle of the road with black expanses on either side) and wrapping in the *V* dimension (to allow the road to repeat off into the distance).

Most rendering APIs (including the book's `Iv` interfaces) support both clamping and wrapping independently in *U* and *V*. In `Iv`, the functions to control texture coordinate "addressing" are `SetAddressingU` and `SetAddressingV`. The road example above would be set up as follows using these interfaces:

```
IvTexture* texture;

// ...

{
   texture->SetAddressingU(kClampTexAddr);
   texture->SetAddressingV(kWrapTexAddr);

   // ...
```

7.11.5 Texture Samplers in Shader Code

Using a texture sampler in shader code is quite simple. As mentioned in section 7.10.4 a fragment shader simply uses a declared texture sampler as an argument to a lookup function. The following shader code declares a texture sampler and uses it along with a set of texture coordinates to determine the fragment color:

```
// GLSL varying vec2 texCoords;
void main() // vertex shader
{
   // Grab the first set of texture coordinates
   // and pass them on
   texCoords = gl_MultiTexCoord0;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

// GLSL - fragment shader
uniform sampler2D texture;
varying vec2 texCoords;

void main()
{
   // Sample the texture represented by "texture"
   // at the location "texCoords"
   gl_FragColor = texture2D (texture, texCoords);
}
```

This is a simple example: The value passed in for the texture coordinate could be computed by other means, either in the vertex shader (and then interpolated automatically as a varying value into the fragment shader), or it could even have been computed in the fragment shader. However, applications should take care to remember that the vertex and fragment shaders are invoked at different frequencies. When possible, it is generally better to put computations that *can* be done in the vertex shader in the vertex shader. If a computation can be done in either the vertex or fragment shader with no difference in visual outcome, it may increase performance to have the shader units compute these values only at each vertex.

7.12 The Steps of Texturing

Unlike basic, per-vertex (Gouraud) shading, texturing adds several levels of indirection between the values defined at the vertices (the *UV* values) and

the final sample colors. This is at once the very power of the method and its most confusing aspect. This indirection means that the colors applied to a triangle by texturing can approximate an extremely complex function, far more complex and detailed than the planar function implied by Gouraud shading. However, it also means that there are far more stages in the method whereupon things can go awry. This section aims to pull together all of the previous texturing discussion into a simple, step-by-step pipeline. Understanding this basic pipeline is key to developing and debugging texturing use in any application.

7.12.1 Other Forms of Texture Coordinates

Real-valued, normalized texture coordinates would *seem* to add a continuity that does not actually exist across the domain of an image, which is a discrete set of color values. For example, in C or C++ one does not access an array with a floating-point value—the index must first be rounded to an integer value. For the purposes of the initial discussion of texturing, we will leave the details of how real-valued texture coordinates map to texture colors somewhat vague. This is actually a rather broad topic and will be discussed in detail in Chapter 9. Initially, it is easiest to think of the texture coordinate as referring to the color of the closest texel. For example, given our assumption, a texture coordinate of $(0.5, 0.5)$ in a texture with width and height equal to 128 texels would map to texel $(64, 64)$. This is referred to as *nearest-neighbor* texture mapping. While this is the simplest method of mapping real-valued texture coordinates into a texture, it is not necessarily the most commonly used in modern applications. We shall discuss more powerful and complex techniques in Chapter 9, but nearest-neighbor mapping is sufficient for the purposes of the initial discussion of texturing.

While normalized texture coordinates are the coordinates that most graphics systems use at the application and shading language level, they are not very useful at all when actually rendering with textures at the lowest level, where we are much more concerned with the texels themselves. We will use them very rarely in the following low-level rendering discussions. We notate normalized texture coordinates simply as (u, v).

The next form of coordinates is often referred to as *texel coordinates*. Like texture coordinates, texel coordinates are represented as real-valued numbers. However, unlike texture coordinates, texel coordinates are dependent upon the width ($w_{texture}$) and height ($h_{texture}$) of the texture image being used. We will notate texel coordinates as (u_{texel}, v_{texel}). The mapping from (u, v) to (u_{texel}, v_{texel}) is

$$(u_{texel}, v_{texel}) = \left(u \cdot w_{texture} - \frac{1}{2}, v \cdot h_{texture} - \frac{1}{2}\right)$$

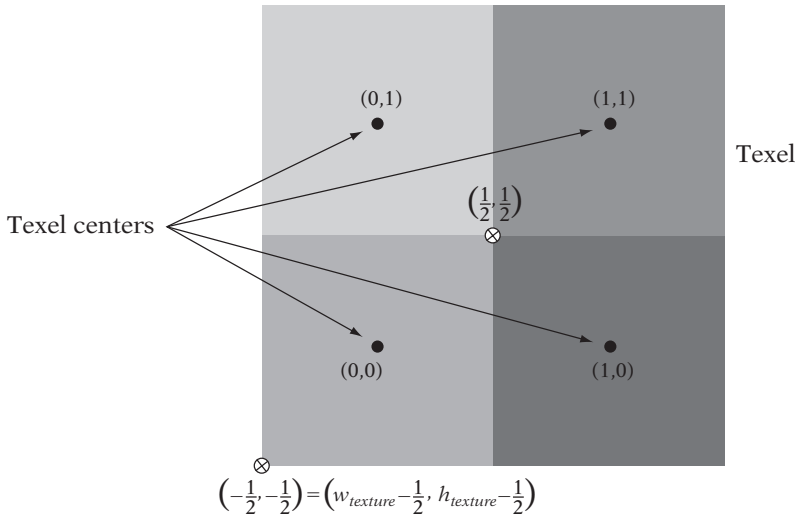


FIGURE 7.20 Texel coordinates and texel centers.

The shift of 1/2 may seem odd, but Figure 7.20 shows why this is necessary. Texel coordinates are relative to the texel *centers*. A texture coordinate of zero is on the boundary between two repetitions of a texture. Since the texel centers are at the middle of a texel, a texture coordinate that falls on an integer value is really halfway between the center of the last texel of one repetition of the texture and the center of the first texel in the next repetition. So a texture coordinate of 0 is equivalent to a *texel* coordinate of −0.5. See [77] (the section "Directly Mapping Texels to Pixels") for details of one common graphics system's texture coordinate to texel mapping.

7.12.2 From Texture Coordinates to a Texture Sample Color

Texturing is a function that maps per-vertex 2-vectors (the texture coordinates), a texture image, and a group of settings into a per-sample color. The top-level stages are as follows:

1. Map the barycentric s and t values into u and v values using the affine mapping defined by the three triangle-vertex texture coordinates: (u_1, v_1), (u_2, v_2), and (u_3, v_3):

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (u_1 - u_3) & (u_2 - u_3) & u_3 \\ (v_1 - v_3) & (v_2 - v_3) & v_3 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

2. Using the texture coordinate mapping mode (either clamping or wrapping), map the U and V values into the unit square:

$$u_{unit}, v_{unit} = wrap(u), wrap(v)$$

or,

$$u_{unit}, v_{unit} = clamp(u), clamp(v)$$

3. Using the width and height of the texture image in texels, map the U and V values into integral texel coordinates via simple scaling:

$$u_{texel}, v_{texel} = \lfloor u_{unit} \times width \rfloor, \lfloor v_{unit} \times height \rfloor$$

4. Using the texture image, map the texel coordinates into colors using image lookup:

$$C_T = Image(u_{texel}, v_{texel})$$

These steps compose to create the mapping from a point on a given triangle to a color value. The following inputs must be configured, regardless of the specific graphics system:

- The texture coordinate being sampled (from interpolated vertex attributes, interpolated from a computation in the vertex shader, or computed in the fragment shader).
- The texture image to be applied.
- The coordinate mapping mode.

7.13 Limitations of Static Shading

The shaders shown in this chapter are about as simple as shaders can possibly be. They project geometry to the screen and directly apply previously assigned vertex colors and textures to a surface. All of the methods described thus far assign colors that do not change for any given sample point at runtime. In other words, no matter what occurs in the scene, a fixed point on a given surface will always return the same color.

Real-world scenes are dynamic, with colors that change in reaction to changes in lighting, position, and even to the surfaces themselves. Any shading method that relies entirely on values that are fixed over both time and

scene conditions will be unable to create truly convincing, dynamic worlds. Methods that can represent real-world lighting and the dynamic nature of moving objects are needed.

Programmable shading is tailor-made for these kinds of applications. A very popular method of achieving these goals is to use a simple, fast approximation of real-world lighting written into vertex and fragment shaders. The next chapter will discuss in detail many aspects of how lighting can be approximated in real-time 3D systems. The chapter will detail more and more complex shaders, adding increasing realism to the rendered scene. The shaders presented will use dynamic inputs, per-vertex and per-pixel math, and textures to simulate the dynamic and complex nature of real-world lighting. Shaders provide an excellent medium for explaining the mathematics of lighting, since in many cases, the mathematical formulae can be directly reflected in shader code. Finally, we will discuss the benefits and issues of computing lighting in the vertex or fragment shaders.

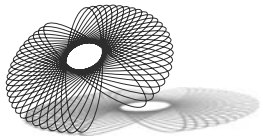
7.14 Chapter Summary

In this chapter we have discussed the basics of procedural shading and the most common inputs to the procedural shading pipeline. These techniques and concepts lay the foundation for the next two chapters, which will discuss popular shading techniques for assigning colors to geometry (dynamic lighting), as well as a detailed discussion of the low-level mathematical issues in computing these colors for display (rasterization). While we have already discussed the basics of the extremely popular shading method known as texturing, this chapter is not the last time we shall mention it. Both of the following two chapters will discuss the ways that texturing affects other stages in the rendering pipeline.

For further reading, popular graphics texts such as Foley et al. [38] detail other aspects of shading, including methods used for high-end offline rendering, which are exactly the kinds of methods that are now starting to be implemented as pixel and vertex shaders in real-time hardware. Shader books such as Engel [31] and Pharr [92] also discuss and provide examples of specific programmable shaders that implement high-end shading methods and can serve as springboards for further experimentation.

This page intentionally left blank

Chapter 8
Lighting



8.1 Introduction

Much of the way we perceive the world visually is based on the way objects in the world react to the light around them. This is especially true when the lighting around us is changing or the lights or objects are moving. Given these facts, it is not surprising that one of the most common uses of programmable shading is to simulate the appearance of real-world lighting.

The coloring methods we have discussed so far have used colors that are statically assigned at content creation time (by the artist) or at the start of the application. These colors do not change on a frame-to-frame basis. At best, these colors represent a "snapshot" of the scene lighting at a given moment for a given configuration of objects. Even if we only intend to model scenes where the lights and objects remain static, these static colors cannot represent the view-dependent nature of lighting with respect to shiny or glossy surfaces.

Clearly, we need a dynamic method of rendering lighting in real time. At the highest level, this requires two basic items: a mathematical model for computing the colors generated by lighting and a high-performance method of implementing this model. We have already introduced the latter requirement; programmable shading pipelines were designed specifically with geometric and color computations (such as lighting) in mind. In this chapter we will greatly expand upon the basic shaders, data sources, and shader syntax that were introduced in Chapter 7. However, we must first address the other requirement — the mathematical model we will use to represent lighting.

The following sections will discuss the details of a popular set of methods for approximating lighting for real-time rendering, as well as examples of how these methods can be implemented as shaders. While we will use

shaders to implement them, the lighting model we will discuss is based upon the long-standing OpenGL fixed-function lighting pipeline (introduced in OpenGL 1.x). At the end of the chapter we will introduce several more advanced lighting techniques that take advantage of the unique abilities of programmable shaders.

We will refer to fixed-function lighting pipelines in many places in this chapter. Fixed-function lighting pipelines were the methods used in rendering application programming interfaces (APIs) to represent lighting calculations prior to the availability of programmable shaders. They are called fixed-function pipelines because the only options available to users of these pipelines were to change the values of predefined colors and settings. The pipelines implemented a basically fixed-structure lighting equation and presented a limited, fixed set of options to the application programmer. No other modifications to the lighting pipeline (and thus the lighting equation or representation) were available. Shaders make it possible to implement the exact lighting methods desired by the particular application.

8.2 Basics of Light Approximation

The physical properties of light are incredibly complex. Even relatively simple scenes never could be rendered realistically without "cheating." In a sense, all of computer graphics is little more than cheating — finding the cheapest-to-compute approximation for a given situation that will still result in a realistic image. Even non-real-time, photorealistic renderings are only approximations of reality, trading off accuracy for ease and speed of computation.

Real-time renderings are even more superficial approximations. Light in the real world reflects, scatters, refracts, and otherwise bounces around the environment. Historically, real-time three-dimensional (3D) lighting often modeled only direct lighting, the light that comes along an unobstructed path from light source to surface. Worse yet, many legacy real-time lighting systems (such as OpenGL and Direct3D's fixed-function lighting pipelines) do not support automatic shadowing. Shadowing involves computing light-blocking effects from objects located between the object being lit and the light source. These are ignored in the name of efficiency. However, despite these limitations, even basic lighting can have a *tremendous* impact on the overall impression of a rendered 3D scene.

Lighting in real-time 3D generally involves data from at least three different sources: the surface configuration (vertex position, normal vector), surface material (how the surface reacts to light), and light emitter properties (the way the light sources emit light). We will discuss each of these sources in terms of how they affect the lighting of an object and will then discuss how these values are passed to the shaders we will be constructing. All of the shader concepts from Chapter 7 (vertex and fragment shading, attributes, uniforms and varying, etc.) will be pivotal in our creation of a lighting system.

8.2.1 Measuring Light

In order to understand the mathematics of lighting, even the simplified, nonphysical approximation used by most real-time 3D systems, it is helpful to know more about how light is actually measured. The simplest way to appreciate how we measure light is in terms of an idealized lightbulb and an idealized surface being lit by that bulb. To explain both the *brightness* and *luminance* (these are actually two different concepts; we will define them in the following section) of a lit surface, we need to measure and track the following path from end to end:

- The amount of light generated by the bulb.
- The amount of light reaching the surface from the bulb.
- The amount of light reaching the viewer from the surface.

Each of these is measured and quantified differently. First, we need a way of measuring the amount of light being generated by the lightbulb. Lightbulbs are generally rated according to several different criteria. The number most people think of with respect to lightbulbs is wattage. For example, we think of a 100-watt lightbulb as being much brighter than a 25-watt lightbulb, and this is generally true when comparing bulbs of the same kind. Wattage in this case is a measure of the electrical power consumed by the bulb in order to create light. It is *not* a direct measure of the amount of light actually *generated* by the bulb. In other words, two lightbulbs may consume the same wattage (say, 100 watts) but produce different amounts of light — one type of bulb simply may be more efficient at converting electricity to light. So what is the measure of light output from the bulb?

Overall light output from a light source is a measure of power: light energy per unit time. This quantity is called *luminous flux*. The unit of luminous flux is the *lumen*. The luminous flux from a lightbulb is measured in lumens, a quantity that is generally listed on boxes of commercially available lightbulbs, near the wattage rating. However, lumens are not how we measure the amount of light that is incident upon a surface.

There are several different ways of measuring the light incident upon a surface. The one that will be of greatest interest to us is *illuminance*. Illuminance is a measure of the amount of luminous flux falling on a given area of surface. Illuminance is also called *luminous flux density*, as it is the amount of luminous flux per unit area. It is measured in units of *lux*, which are defined as lumens per meter squared. Illuminance is an important quantity because it measures not only the light power (in lumens), but also the area over which this power is distributed (in square meters). Given a fixed amount of luminous flux, increasing the surface area over which it is distributed will decrease the illuminance proportionally. We will see this property again later, when we

discuss the illuminance from a point light source. Illuminance in this case is only the light incident upon a surface, *not* the amount reflected from the surface.

Light reflection from a surface depends on a lot of properties of the surface and the geometric configuration. We will cover approximations of reflection later in this chapter. However, the final step in our list of lighting measurements is to define how we measure the reflected light reaching the viewer from the surface. The quantity used to measure this is *luminance*, which is defined as illuminance per unit solid angle. Luminance thus takes into account how the reflected light is spread directionally. The unit of luminance is the *nit*, and this value is the closest of those we have discussed to representing "brightness." However, brightness is a perceived value and is not linear with respect to luminance, due to the response curve of the human visual system. For details of the relationship between brightness and luminance, see Cornsweet [20].

The preceding quantities are photometric; that is, they are weighted by the human eye's response to different wavelengths of light. The field of radiometry studies the measurement of analogous quantities that do not include this physiological weighting. The radiometric equivalent of illuminance is *irradiance* (measured in watts per meter squared), and the equivalent of luminance is *radiance*. These radiometric units and quantities are relevant to anyone working with computer graphics, as they are commonly seen in the field of non-real-time rendering, especially in techniques known collectively as *global illumination* (see Cohen and Wallace [19]).

8.2.2 Light as a Ray

Our discussion of light sources will treat light from a light source as a collection of rays, or in some cases simply as vectors. These rays represent infinitely narrow "shafts" of light. This representation of light will make it much simpler to approximate light–surface interaction. Our light rays will often have RGB (red, green, blue) colors or scalars associated with them that represent the intensity (and in the case of RGB values, the color) of the light incident upon a surface. While this value is often described in rendering literature as "brightness" or even "luminance," these terms are descriptive rather than physically based. In fact, these intensity values are more closely related to and roughly approximate the *illuminance* incident upon the given surface from the light source.

8.3 A Simple Approximation of Lighting

For the purposes of introducing a real-time lighting equation, we will start by discussing an approximation that is based on OpenGL's original fixed-function lighting model (or pipeline); Direct3D's original fixed-function

lighting pipeline was similar. Initially, we will speak in terms of lighting a "sample": a generic point in space that may represent a vertex in a tessellation or a fragment in a triangle. We will attempt to avoid the concepts of vertices and fragments during this initial discussion, preferring to refer to a general point on a surface, along with a local surface normal and a surface material. (As will be detailed later, a surface material contains all of the information needed to determine how an object's surface reacts to lighting.) Once we have introduced the concepts, however, we will discuss how vertex and fragment shaders can be used to implement this model, along with the trade-offs of implementing it in one shading unit or another. As already mentioned, this simple lighting model does not accurately represent the real world — there are many simplifications required for real-time lighting performance.

While OpenGL and Direct3D (prior to DX10) support fixed-function lighting pipelines, and can even pass light and material information down to the shaders from these existing fixed-function interfaces, we will avoid using any parts of the OpenGL fixed-function interfaces. We will instead use custom uniforms for passing down this information to the shader. This allows our discussion to be more easily applied to Direct3D's HLSL shaders (whose fixed-function interfaces differ from OpenGL) and OpenGL ES's GLSL-E (which does not include any fixed-function pipeline).

8.4 Types of Light Sources

The next few sections will discuss the common types of light sources that appear in real-time 3D systems. Each section will open with a general discussion of a given light source, followed by coverage in mathematical terms, and close with the specifics of implementation in shader code (along with a description of the accompanying C code to feed the required data to the shader). The discussion will progress (roughly) from the simplest (and least computationally expensive) light sources to the most complex. Initially, we will look at one light source at a time, but will later discuss how to implement multiple simultaneous light sources.

For each type of light source, we will be computing two important values: the unit vector $\hat{\mathbf{L}}$ (here, we break with our notational convention of lowercase vectors in order to make the equations more readable) and the scalar i_L. The vector $\hat{\mathbf{L}}$ is the *light direction vector* — it points from the current surface sample point P_V *toward* the source of the light.

The scalar i_L is the *light intensity value*, which is a rough approximation of the illuminance from the light source at the given surface location P_V. With some types of lights, there will be per-light *tuning* values that adjust the function that defines i_L. In addition, in each of the final lighting term equations, we will also multiply i_L by RGB colors that adjust this overall light intensity

value. These color terms are of the form L_A, L_D, and so on. They will be defined per light and per lighting component and will (in a sense) approximate a scale factor upon the overall luminous flux from the light source.

The values $\hat{\mathbf{L}}$ and i_L do not take any information about the surface orientation or material itself into account, only the relative positions of the light source and the sample point with respect to each other. Discussion of the contribution of surface orientation (i.e., the surface normal) will be taken up later, as each type of light and component of the lighting equation will be handled differently and independent of the particular light source type.

8.4.1 Directional Lights

SOURCE CODE DEMO DirectionalLight

A directional light source (also known as an infinite light source) is similar to the light of the Sun as seen from Earth. Relative to the size of the Earth, the Sun seems almost infinitely far away, meaning that the rays of light reaching Earth from the Sun are basically parallel to one another, independent of position on Earth. Consider the source and the light it produces as a single vector. A directional light is defined by a *point at infinity*, P_L. The light source direction is produced by turning the point into a unit vector (by subtracting the position of the origin and normalizing the result):

$$\hat{\mathbf{L}} = \frac{P_L - 0}{|P_L - 0|}$$

Figure 8.1 shows the basic geometry of a directional light. Note that the light rays are the negative (reverse) of the light direction vector $\hat{\mathbf{L}}$, since $\hat{\mathbf{L}}$ points from the surface to the light source.

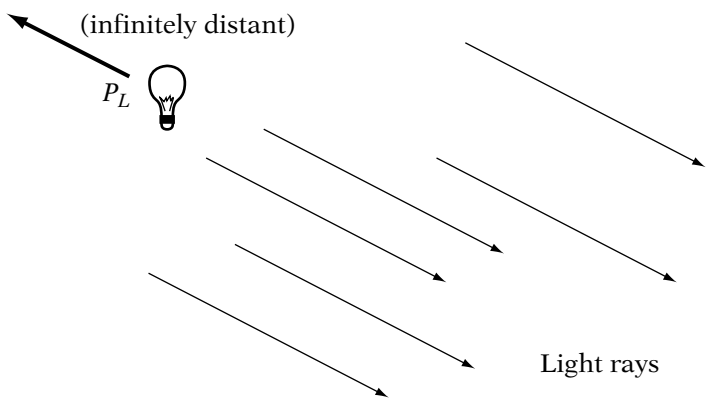


Figure 8.1 The basic geometry of a directional light.

The value i_L for a directional light is constant for all sample positions:

$$i_L = 1$$

Since both i_L and light vector $\hat{\mathbf{L}}$ are constant for a given light (and independent of the sample point P_V), directional lights are the least computationally expensive type of light source. Neither $\hat{\mathbf{L}}$ nor i_L needs to be recomputed for each sample. As a result, we will pass both of these values to the shader (vertex or fragment) as uniforms and use them directly. We define a standard structure in GLSL code to hold the i_L and $\hat{\mathbf{L}}$ values.

```
struct lightSampleValues {
   vec3 L;
   float iL;
};
```

And we define a function for each type of light that will return this structure.

```
// GLSL Code

// normalized vector with z == 0
uniform vec4 dirLightPosition;
uniform float dirLightIntensity;

// Later, in the code, we can use these values directly...
lightSampleValues computeDirLightValues()
{
   lightSampleValues values;
   values.L = dirLightPosition.xyz;
   values.iL = dirLightIntensity;
   return values;
}
```

8.4.2 Point Lights

A point or positional light source (also known as a local light source to differentiate it from an infinite source) is similar to a bare lightbulb, hanging in space. It illuminates equally in all directions. A point light source is defined by its location, the point P_L. The light source direction produced is

$$\hat{\mathbf{L}} = \frac{P_L - P_V}{|P_L - P_V|}$$

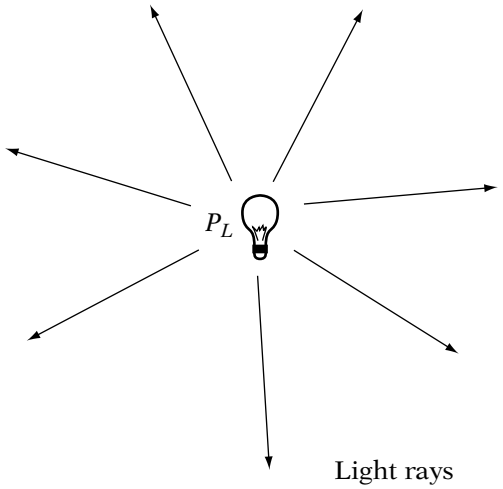


FIGURE 8.2 The basic geometry of a point light.

This is the normalized vector that is the difference from the sample position to the light source position. It is not constant per-sample, but rather forms a vector field that points toward P_L from all points in space. This normalization operation is one factor that often makes point lights more computationally expensive than directional lights. While this is not a prohibitively expensive operation to compute *once per light*, we must compute the subtraction of two points and normalize the result to compute this light vector for *each lighting sample* (generally per vertex for each light) for every frame. Figure 8.2 shows the basic geometry of a point light.

We specify the location of a point light in the same space as the vertices (normally view space, for reasons that will be discussed later in this section) using a 4-vector with a nonzero w coordinate. The position of the light can be passed down as a uniform to the shader, but note that we cannot use that position directly as $\hat{\mathbf{L}}$. We must compute the value of $\hat{\mathbf{L}}$ per sample using the position of the current sample, which we will define to be the 4-vector `surfacePosition`. In a vertex shader, this would be the vertex position attribute transformed into view space, while in the fragment shader, it would be an interpolated varying value representing the surface position in view space at the sample.

```
// GLSL Code
uniform vec4 pointLightPosition; // position with w == 1

// Later, in the code, we must compute L per sample...
// as described above, surfacePosition is passed in from a
```

```
// per-vertex attribute or a per-sample varying value
lightSampleValues computePointLightValues(in vec4 surfacePosition)
{
   lightSampleValues values;
   values.L = normalize(pointLightPosition - surfacePosition).xyz;
   // we will add the computation of values.iL later

   return values;
}
```

Unlike a directional light, a point light has a nonconstant function defining i_L. This nonconstant intensity function approximates a basic physical property of light known as the *inverse-square law*: Our idealized point light source radiates a constant amount of luminous flux, which we call I, at all times. In addition, this light power is evenly distributed in all directions from the point source's location. Thus, any cone-shaped subset (a *solid angle*) of the light coming from the point source represents a constant fraction of this luminous flux (we will call this I_{cone}). An example of this conical subset of the sphere is shown in Figure 8.3.

Illuminance (the photometric value most closely related to our i_L) is measured as luminous flux per unit area. If we intersect the cone of light with a plane perpendicular to the cone, the intersection forms a disc (see Figure 8.3). This disc is the surface area illuminated by the cone of light. If we assume that this plane is at a distance *dist* from the light center and the radius of the resulting disc is r, then the area of the disc is πr^2. The illuminance E_{dist} (in the literature, illuminance is generally represented with the letter E) is proportional to

$$E_{dist} = \frac{power}{area} \propto \frac{I_{cone}}{\pi r^2}$$

However, at a distance of $2dist$, then the radius of the disc is $2r$ (see Figure 8.3). The resulting radius is $\pi(2r)^2$, giving an illuminance E_{2dist} proportional to

$$E_{2dist} \approx \frac{I_{cone}}{\pi(2r)^2} = \frac{I_{cone}}{4\pi r^2} = \frac{E_{dist}}{4}$$

Doubling the distance divides (or attenuates) the illuminance by a factor of four, because the same amount of light energy is spread over four times the surface area. This is known as the inverse-square law (or more generally as *distance attenuation*), and it states that for a point source, the illuminance decreases with the square of the distance from the source. As an example of a practical application, the inverse-square law is the reason why a candle can illuminate a small room that is otherwise completely unlit but will *not*

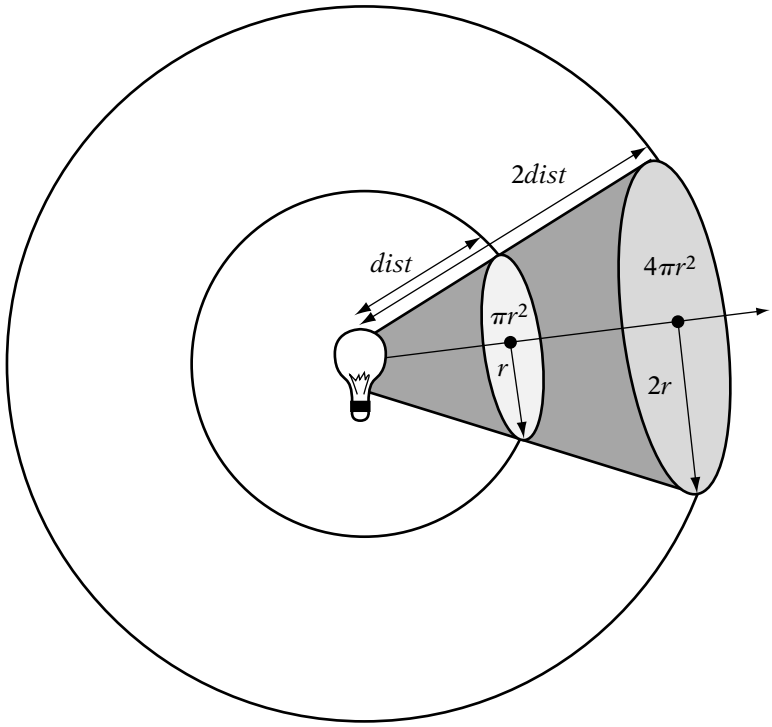


FIGURE 8.3 The inverse-square law.

illuminate an entire stadium. In both cases, the candle provides the same amount of luminous flux. However, the actual surface areas that must be illuminated in the two cases are vastly different due to distance.

The inverse-square law results in a basic i_L for a point light equal to

$$i_L = \frac{1}{dist^2}$$

where

$$dist = |P_L - P_V|$$

which is the distance between the light position and the sample position.

While exact inverse-square law attenuation is physically correct, it does not always work well artistically or perceptually. As a result, OpenGL and most other fixed-function and shader-based lighting pipelines support a more general distance attenuation function for point lights: a general quadratic.

Under such a system, the function i_L for a point light is

$$i_L = \frac{1}{k_c + k_l dist + k_q dist^2}$$

The distance attenuation constants k_c, k_l, and k_q are defined per light and determine the shape of that light's attenuation curve. Figure 8.4 is a visual

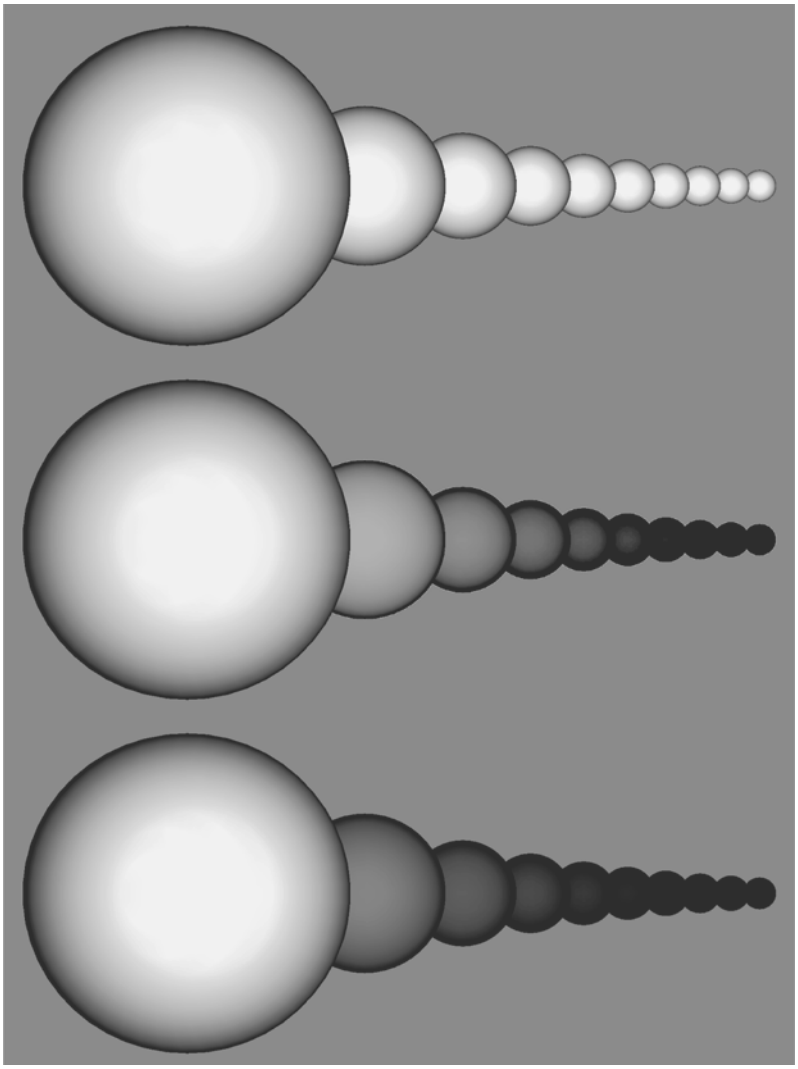


FIGURE 8.4 Distance attenuation.

example of constant, linear, and quadratic attenuation curves. The spheres in each row increase in distance linearly from left to right.

Generally, *dist* should be computed in "eye" or camera coordinates (post–model view transform); this specification of the space used is important, as there may be scaling differences between model space, world space, and camera space, which would change the scale of the attenuation. Most importantly, model-space scaling differs per object, meaning the different objects whose model transforms have different scale would be affected differently by distance attenuation. This would not look correct. Distance attenuation must occur in a space that uses the same scale factor for all objects in a scene. The three distance attenuation values can be passed down as a single 3-vector uniform, with the x, y, and z components containing k_c, k_l, and k_q, respectively. Since the attenuation must be computed per sample and involves the length of the $P_L - P_V$ vector, we merge the i_L shader code into the previous $\hat{\mathbf{L}}$ shader code as follows:

```
// GLSL Code
uniform vec4 pointLightPosition; // position with w == 1
uniform float pointLightIntensity;
uniform vec3 pointLightAttenuation; // (k_c, k_l, k_q)

lightSampleValues computePointLightValues(in vec4 surfacePosition)
{
   lightSampleValues values;
   values.L = pointLightPosition.xyz - surfacePosition.xyz;
   float dist = length(values.L);
   values.L = values.L / dist; // normalize

   // Dot computes the 3-term attenuation in one operation
   // k_c * 1.0 + k_l * dist + k_q * dist * dist
   float distAtten = dot(pointLightAttenuation,
                         vec3(1.0, dist, dist*dist));
   values.iL = pointLightIntensity / distAtten;

   return values;
}
```

The attenuation of a point light's intensity by this quadratic can be computationally expensive, as it must be recomputed per sample. In order to increase performance on some systems, shaders sometimes leave out one or more terms of the distance attenuation equation entirely.

8.4.3 Spotlights

SOURCE CODE DEMO SpotLight

A spotlight is like a point light source with the ability to limit its light to a cone-shaped region of the world. The behavior is similar to a theatrical spotlight with the ability to focus its light on a specific part of the scene.

In addition to the position P_L that defined a point light source, a spotlight is defined by a direction vector $\mathbf{d}$, a scalar cone angle θ, and a scalar exponent s. These additional values define the direction of the cone and the behavior of the light source as the sample point moves away from the central axis of the cone. The infinite cone of light generated by the spotlight has its apex at the light center P_L, an axis $\mathbf{d}$ (pointing *toward the base* of the cone), and a half angle of θ. Figure 8.5 illustrates this configuration. The exponent s is not a part of the geometric cone; as will be seen shortly, it is used to attenuate the light within the cone itself.

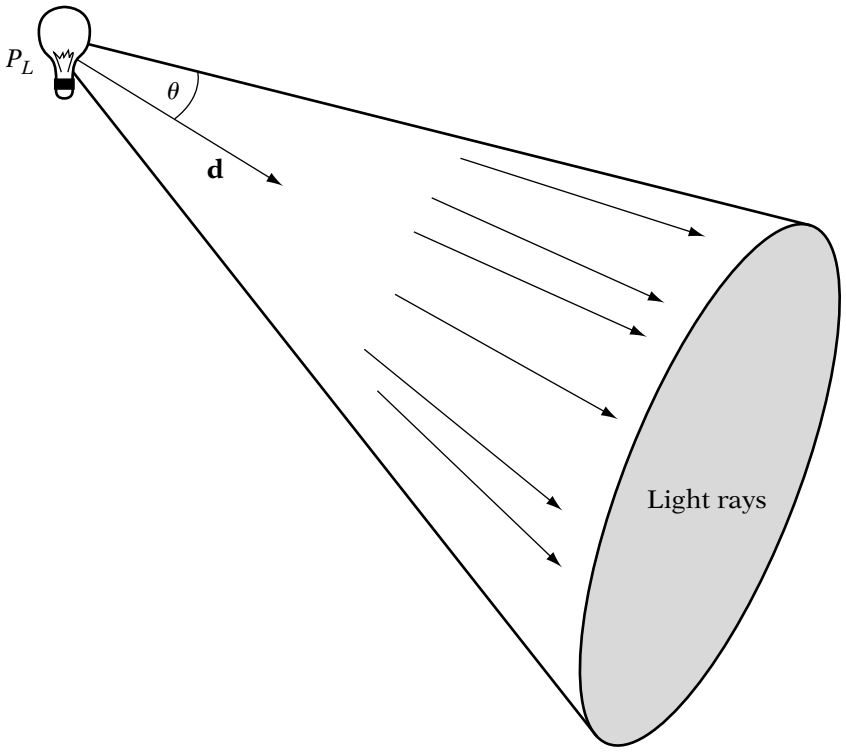


Figure 8.5 The basic geometry of a spotlight.

The light vector is equivalent to that of a point light source:

$$\hat{\mathbf{L}} = \frac{P_L - P_V}{|P_L - P_V|}$$

For a spotlight, i_L is based on the point light function but adds an additional term to represent the focused, conical nature of the light emitted by a spotlight:

$$i_L = \frac{spot}{k_c + k_l dist + k_q dist^2}$$

where

$$spot = \begin{cases} (-\hat{\mathbf{L}} \cdot \mathbf{d})^s, & \text{if } (-\hat{\mathbf{L}} \cdot \mathbf{d}) \geq \cos\theta \\ 0, & \text{otherwise} \end{cases}$$

As can be seen, the *spot* term is 0 when the sample point is outside of the cone. The *spot* term makes use of the fact that the light vector and the cone vector are normalized, causing $(-\hat{\mathbf{L}} \cdot \mathbf{d})$ to be equal to the cosine of the angle between the vectors. We must negate $\hat{\mathbf{L}}$ because it points toward the light, while the cone direction vector $\mathbf{d}$ points away from the light. Computing the cone term first can allow for performance improvements by skipping the rest of the light calculations if the sample point is outside of the cone. In fact, some graphics systems even check the bounding volume of an object against the light cone, avoiding any spotlight computation on a per-sample basis if the object is entirely outside of the light cone.

Inside of the cone, the light is attenuated via a function that does not represent any physical property but is designed to allow artistic adjustment. The light's i_L function reaches its maximum inside the cone when the vertex is along the ray formed by the light location P_L and the direction $\mathbf{d}$, and decreases as the vertex moves toward the edge of the cone. The dot product is used again, meaning that i_L falls off proportionally to

$$\cos^s \omega$$

where ω is the angle between the cone direction vector and the vector between the sample position and the light location $(P_V - P_L)$. As a result, the light need not attenuate smoothly to the cone edge — there may be a sharp drop to $i_L = 0$ right at the cone edge. Adjusting the s value will change the rate at which i_L falls to 0 inside the cone as the sample position moves off axis.

The multiplication of the *spot* term with the distance attenuation term means that the spotlight will attenuate over distance within the cone. In this way, it acts exactly like a point light with an added conic focus. The fact that

both of these expensive attenuation terms must be recomputed per sample makes the spotlight the most computationally expensive type of standard light in most systems. When possible, applications attempt to minimize the number of simultaneous spotlights (or even avoid their use altogether).

Spotlights with circular attenuation patterns are not universal. Another popular type of spotlight (see Warn [116]) models the so-called barn door spotlights that are used in theater, film, and television. However, because of these additional computational expenses, conical spotlights are by far the more common form in real-time graphics systems.

As described previously, $\hat{\mathbf{L}}$ for a spotlight is computed as for a point light. In addition, the computation of i_L is similar, adding an additional term for the spotlight angle attenuation. The spotlight-specific attenuation requires several new uniform values per light, specifically:

- `spotLightDir`: A unit-length 3-vector representing the spotlight direction.
- `spotLightAngleCos`: The cosine of the half-angle of the spotlight's cone.
- `spotLightExponent`: The exponent used to adjust the cone attenuation.

These values and the previous formulae are then folded into the earlier shader code for a point light, giving the following computations:

```
// GLSL Code
uniform vec4 spotLightPosition; // position with w == 1
uniform float spotLightIntensity;
uniform vec3 spotLightAttenuation; // (k_c, k_l, k_q)
uniform vec3 spotLightDir; // unit-length
uniform float spotLightAngleCos;
uniform float spotLightExponent;


lightSampleValues computeSpotLightValues(in vec4 surfacePosition)
{
   lightSampleValues values;
   values.L = spotLightPosition.xyz - surfacePosition.xyz;
   float dist = length(values.L);
   values.L = values.L / dist; // normalize

   // Dot computes the 3-term attenuation in one operation
   // k_c * 1.0 + k_l * dist + k_q * dist * dist
   float distAtten = dot(spotLightAttenuation,
                         vec3(1.0, dist, dist*dist));

   float spotAtten = dot(-spotLightDir, values.L);
```

```
    spotAtten = (spotAtten > spotLightAngleCos)
                   ? pow(spotAtten, spotLightExponent)
                   : 0.0;

    values.iL = spotLightIntensity * spotAtten / distAtten;

    return values;
}
```

8.4.4 Other Types of Light Sources

The light sources above are only a few of the most basic that are seen in modern lighting pipelines, although they serve the purpose of introducing shader-based lighting quite well. There are many other forms of lights that are used in shader-based pipelines. We will discuss several of these at a high level and provide more detailed references in the advanced lighting sections at the end of the chapter.

One thing all of the light sources in the previous sections have in common is that a single vector can represent the direct lighting from each source at a particular sample on a surface. The lights described thus far are either infinitely distant or emit from a single point. Lights in the real world very often emit light not from a single point, but from a larger area. For example, the diffused fluorescent light fixtures that are ubiquitous in office buildings appear to emit light from a large, rectangular surface. There are two basic effects produced by these area light sources that are not represented by any of our lights above: a solid angle of incoming light upon the surface, and soft shadows.

One aspect of area light sources is that the direct lighting from them that is incident upon a single point on a surface comes from multiple directions. In fact, the light from an area light source on a surface point forms a complex, roughly cone-shaped volume whose apex is at the surface point being lit. Unless the area of the light source is large relative to its distance to the surface, the effect of this light coming from a range of directions can be very subtle. As the ratio of the area of the light source to the distance to the object (the projected size of the light source from the point of view of the surface point) goes down, the effect can rapidly converge to look like the single-vector cases we describe above.

The main interest in area light sources has to do with occlusion of the light from them, namely the soft-edged shadows that this partial occlusion produces. This effect can be very significant, even if the area of the light source is quite small. Soft-edged shadows occur at shadow boundaries, where the point in partial shadow is illuminated by part of the area light source but not all of it. The shadow becomes progressively darker as the given surface point can "see" less and less of the area light source. This soft shadow region (called

the *penumbra*, as opposed to the fully shadowed region, called the *umbra*) is highly prized in non-real-time, photorealistic renderings for the realistic quality it lends to the results.

Soft shadows and other area light effects are not generally supported in low-level, real-time 3D graphics software development kits (SDKs) (including OpenGL). However, high-level rendering engines based upon programmable shaders are implementing these effects in a number of ways in modern applications. The advanced lighting sections at the end of this chapter describe and reference a few of these methods. However, our introduction will continue to discuss the light incident upon a surface from a given light source with a single vector.

8.5 Surface Materials and Light Interaction

Having discussed the various ways in which the light sources in our model generate light incident upon a surface, we must complete the model by discussing how this incoming light (our approximation of illuminance) is converted (or reflected) into outgoing light (our approximation of luminance) as seen by the viewer or camera. This section will discuss a common real-time model of light–surface reflection.

In the presence of lighting, there is more to surface appearance than a single color. Surfaces respond differently to light, depending upon their composition, for example, unfinished wood, plastic, or metal. Gold-colored plastic, gold-stained wood, and actual gold all respond differently to light, even if they are all the same basic color. Most real-time 3D lighting models take these differences into account with the concept of a material.

A material describes the behavior of an object with respect to light. In our real-time rendering model, a material describes the way a surface generates or responds to four different categories of light: emitted light, ambient light, diffuse light, and specular light. Each of these forms of light is an approximation of real-world light, and, put together, they can serve well at differentiating not only the colors of surfaces but also the apparent compositions (shiny versus matte, plastic versus metal, etc.). Each of the four categories of approximated light will be individually discussed.

As with the rest of the chapter, the focus will be on a lighting model similar to the one that is used by OpenGL and Direct3D's fixed-function pipelines. Most of these concepts carry over to other common low-level, real-time 3D SDKs as well, even if the methods of declaring these values and the exact interaction semantics might differ slightly from API to API. We will represent the surface material properties of an object using shader uniform values.

For our lighting model, we will define four colors for each material and one color for each lighting component. These will be defined in each of the following sections.

We will define only one color and one vector for each light: the color of the light, a 3-vector uniform `lightColor`, and a vector whose components represent scalar scaling values of that color per lighting component. This 3-vector will store the scaling factor for each applicable lighting category in a different vector component. We will call this uniform 3-vector `lightAmbDiffSpec`.

8.6 Categories of Light

8.6.1 Emission

Emission, or emissive light, is the light produced by the surface itself, in the absence of any light sources. Put simply, it is the color and intensity with which the object "glows." Because this is purely a surface-based property, only surface materials (not lights) contain emissive colors. The emissive color of a material is written as M_E. One approximation that is made in real-time systems is the (sometimes confusing) fact that this "emitted" light does not illuminate the surfaces of any other objects. In fact, another common (and perhaps more descriptive) term used for emission is *self-illumination*. The fact that emissive objects do not illuminate one another avoids the need for the graphics systems to take other objects into account when computing the light at a given point.

We will store the emissive color of an object's material (M_E) in the 3-vector shader uniform value `materialEmissive`.

8.6.2 Ambient

Ambient light is the term used in real-time lighting as an umbrella under which all forms of indirect lighting are grouped and approximated. Indirect lighting is light that is incident upon a surface not via a direct ray from light to surface, but rather via some other, more complex path. In the real world, light can be scattered by particles in the air, and light can "bounce" multiple times around a scene prior to reaching a given surface. Accounting for these multiple bounces and random scattering effects is very difficult if not impossible to do in a real-time rendering system, so most systems use a per-light, per-material constant for all ambient light.

A light's ambient color represents the color and intensity of the light from a given source that is to be scattered through the scene. The ambient material color represents how much of the overall ambient light the particular surface reflects.

Ambient light has no direction associated with it. However, most lighting models do attenuate the ambient light from each source based on the light's intensity function at the given point, i_L. As a result, point and spotlights do not produce equal amounts of ambient light throughout the scene. This tends to localize the ambient contribution of point and spotlights spatially and keeps ambient light from overwhelming a scene. The overall ambient term for a given light and material is thus

$$C_A = i_L L_A M_A$$

where L_A is the light's ambient color, and M_A is the material's ambient color. Figure 8.6 provides a visual example of a sphere lit by purely ambient light. Without any ambient lighting, most scenes will require the addition of many lights to avoid dark areas, leading to decreased performance. Adding some ambient light allows specific light sources to be used more artistically, to highlight parts of the scene that can benefit from the added dimension of

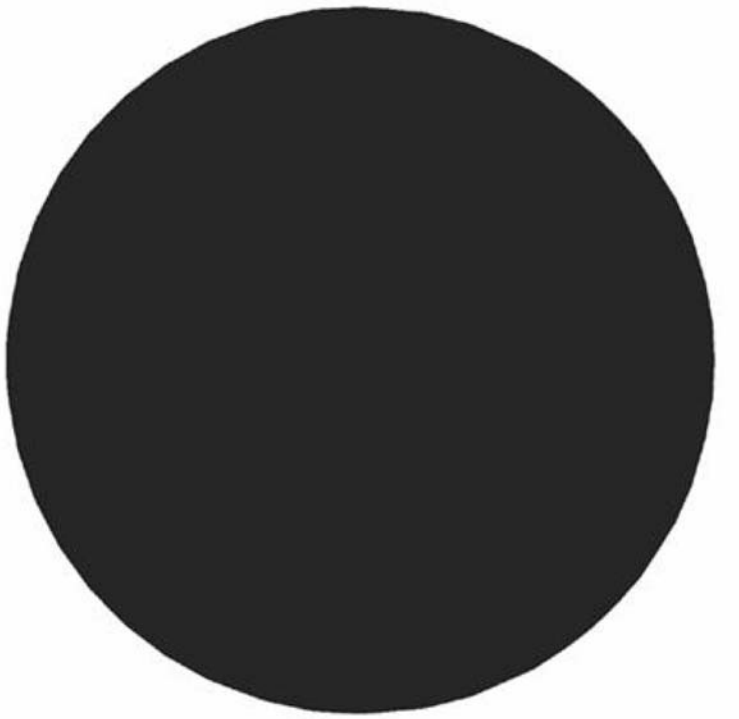


Figure 8.6 Sphere lit by ambient light.

dynamic lighting. However, adding too much ambient light can lead to the scene looking "flat," as the ambient lighting dominates the coloring.

We will store the ambient color of an object's material in the 3-vector shader uniform value `materialAmbientColor`. We will compute the ambient component of a light by multiplying a scalar ambient light factor, `lightAmbDiffSpec.x` (we store the ambient scaling factor in the x component of the vector), times the light color, giving (`lightColor * lightAmbDiffSpec.x`). The shader code to compute the ambient component is as follows:

```
// GLSL Code
uniform vec3 materialAmbientColor;
uniform vec3 lightAmbDiffSpec;
uniform vec3 lightColor;

vec3 computeAmbientComponent(in lightSampleValues light)
{
   return light.iL * (lightColor * lightAmbDiffSpec.x)
                   * materialAmbientColor;
}
```

8.6.3 Diffuse

Diffuse lighting, unlike the previously discussed emissive and ambient terms, represents direct lighting. The diffuse term is dependent on the lighting incident upon a point on a surface from each single light via the direct path. As such, diffuse lighting is dependent on material colors, light colors, i_L, *and* the vectors $\hat{\mathbf{L}}$ and $\hat{\mathbf{n}}$.

The diffuse lighting term treats the surface as a pure diffuse (or matte) surface, sometimes called a *Lambertian reflector*. These surfaces have the property that their luminance is independent of view direction. In other words, like our earlier approximation terms, emissive and ambient, the diffuse term is not view-dependent. The luminance is dependent on only the incident illuminance.

The illuminance incident upon a surface is proportional to the luminous flux incident upon the surface, divided by the surface area over which it is distributed. In our earlier discussion of illuminance, we assumed (implicitly) that the surface in question was perpendicular to the light direction. If we define an infinitesimally narrow ray of light with direction $\hat{\mathbf{L}}$ to have luminous flux I and cross-sectional area δa (Figure 8.7), then the illuminance E incident upon a surface whose normal $\hat{\mathbf{n}} = \hat{\mathbf{L}}$ is

$$E \propto \frac{I}{\delta a}$$

However, if $\hat{\mathbf{n}} \neq \hat{\mathbf{L}}$ (i.e., the surface is not perpendicular to the ray of light), then the configuration is as shown in Figure 8.8. The surface area

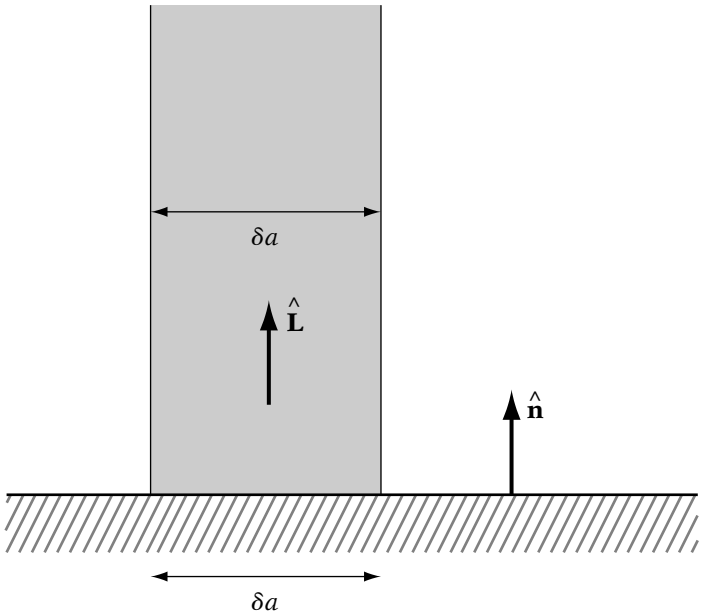


FIGURE 8.7 A shaft of light striking a perpendicular surface.

intersected by the (now oblique) ray of light is represented by $\delta a'$. From basic trigonometry and Figure 8.8, we can see that

$$\begin{aligned}\delta a' &= \frac{\delta a}{\sin\left(\frac{\pi}{2} - \theta\right)} \\ &= \frac{\delta a}{\cos\theta} \\ &= \frac{\delta a}{\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}}\end{aligned}$$

And, we can compute the illuminance E' as follows:

$$\begin{aligned}E' &\propto \frac{I}{\delta a'} \\ &\propto I\left(\frac{\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}}{\delta a}\right) \\ &\propto \left(\frac{I}{\delta a}\right)(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \\ &\propto E(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\end{aligned}$$

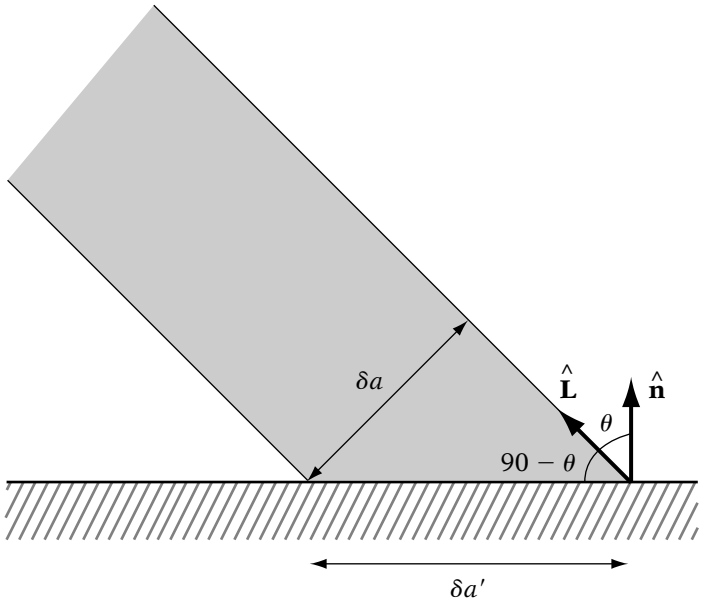


FIGURE 8.8 The same shaft of light at a glancing angle.

Note that if we evaluate for the original special case $\hat{\mathbf{n}} = \hat{\mathbf{L}}$, the result is $E' = E$, as expected. Thus, the reflected diffuse luminance is proportional to $(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})$. Figure 8.9 provides a visual example of a sphere lit by a single light source that involves only diffuse lighting.

Generally, both the material and the light include diffuse color values (M_D and L_D, respectively). The resulting diffuse color for a point on a surface and a light is then equal to

$$C_D = i_L max(0, \hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) L_D M_D$$

Note the *max*() function that clamps the result to 0. If the light source is behind the surface (i.e., $\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} < 0$), then we assume that the back side of the surface obscures the light (self-shadowing), and no diffuse lighting occurs.

We will store the diffuse color of an object's material in the 4-vector shader uniform value `materialDiffuseColor`. The diffuse material color is a 4-vector because it includes the alpha component of the surface as a whole. We will compute the diffuse component of a light by multiplying a scalar ambient light factor, `lightAmbDiffSpec.y` (we store the diffuse scaling factor in the *y* component of the vector), times the light color, giving (`lightColor * lightAmbDiffSpec.y`).

The shader code to compute the diffuse component is as follows. Note that adding the suffix `.rgb` to the end of a 4-vector creates a 3-vector out

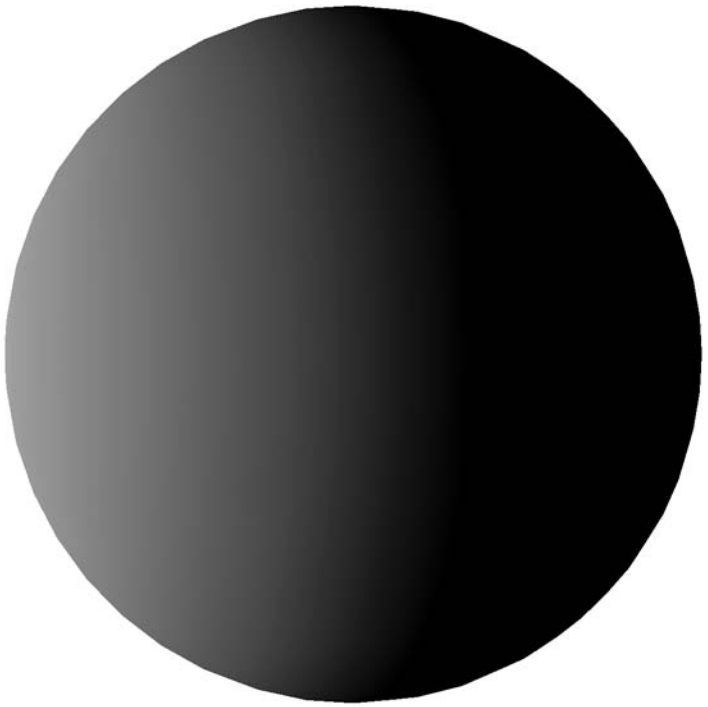


FIGURE 8.9 Sphere lit by diffuse light.

of the red, green, and blue components of the 4-vector. We assume that the surface normal vector at the sample point, $\hat{\mathbf{n}}$, is passed into the function. This value may either be a per-vertex attribute in the vertex shader, an interpolated varying value in the fragment shader, or perhaps even computed in either shader. The source of the normal is unimportant to this calculation.

```
// GLSL Code
uniform vec3 materialDiffuseColor;
uniform vec3 lightAmbDiffSpec;
uniform vec3 lightColor;

// surfaceNormal is assumed to be unit-length
vec3 computeDiffuseComponent(in vec3 surfaceNormal,
                             in lightSampleValues light)
{
   return light.iL * (lightColor * lightAmbDiffSpec.y)
                   *  materialDiffuseColor.rgb
                   *  max(0.0, dot(surfaceNormal, light.L));
}
```

8.6.4 Specular

A perfectly smooth mirror reflects all of the light from a given direction $\hat{\mathbf{L}}$ out along a single direction, the reflection direction $\hat{\mathbf{r}}$. While few surfaces approach completely mirrorlike behavior, most surfaces have at least some mirrorlike component to their lighting behavior. As a surface becomes rougher (at a microscopic scale), it no longer reflects all light from $\hat{\mathbf{L}}$ out along a single direction $\hat{\mathbf{r}}$, but rather in a distribution of directions centered about $\hat{\mathbf{r}}$. This tight (but smoothly attenuating) distribution around $\hat{\mathbf{r}}$ is often called a *specular highlight* and is often seen in the real world. A classic example is the bright white "highlight" reflections seen on smooth, rounded plastic objects. The specular component of real-time lighting is an entirely empirical approximation of this reflection distribution, specifically designed to generate these highlights.

Because specular reflection represents mirrorlike behavior, the intensity of the term is dependent on the relative directions of the light ($\hat{\mathbf{L}}$), the surface normal ($\hat{\mathbf{n}}$), *and* the viewer ($\hat{\mathbf{v}}$). Prior to discussing the specular term itself, we must introduce the concept of the light reflection vector $\hat{\mathbf{r}}$. Computing the reflection of a light vector $\hat{\mathbf{L}}$ about a plane normal $\hat{\mathbf{n}}$ involves negating the component of $\hat{\mathbf{L}}$ that is perpendicular to $\hat{\mathbf{n}}$. We do this by representing $\hat{\mathbf{L}}$ as the weighted sum of $\hat{\mathbf{n}}$ and a unit vector $\hat{\mathbf{p}}$ that is perpendicular to $\hat{\mathbf{n}}$ (but in the plane defined by $\hat{\mathbf{n}}$ and $\hat{\mathbf{L}}$) as follows and as depicted in Figure 8.10:

$$\hat{\mathbf{L}} = l_n \hat{\mathbf{n}} + l_p \hat{\mathbf{p}}$$

The reflection of $\hat{\mathbf{L}}$ about $\hat{\mathbf{n}}$ is then

$$\hat{\mathbf{r}} = l_n \hat{\mathbf{n}} - l_p \hat{\mathbf{p}}$$

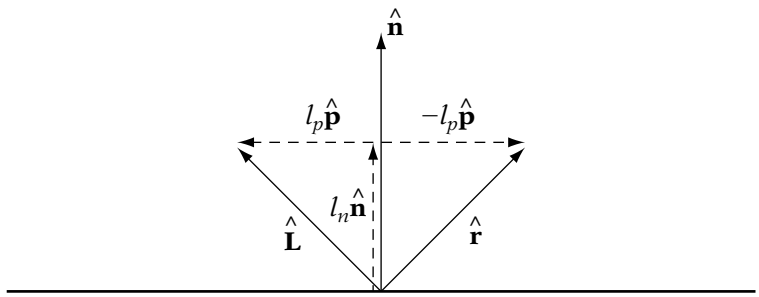


Figure 8.10 The relationship between the surface normal, light direction, and the reflection vector.

We know that the component of $\hat{\mathbf{L}}$ in the direction of $\hat{\mathbf{n}}$ (l_n) is the projection of $\hat{\mathbf{L}}$ onto $\hat{\mathbf{n}}$, or

$$l_n = \hat{\mathbf{L}} \cdot \hat{\mathbf{n}}$$

Now we can compute $l_p \hat{\mathbf{p}}$ by substitution of our value for l_n:

$$\begin{aligned} \hat{\mathbf{L}} &= l_n \hat{\mathbf{n}} + l_p \hat{\mathbf{p}} \\ \hat{\mathbf{L}} &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} + l_p \hat{\mathbf{p}} \\ l_p \hat{\mathbf{p}} &= \hat{\mathbf{L}} - (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} \end{aligned}$$

So, the reflection vector $\hat{\mathbf{r}}$ equals

$$\begin{aligned} \hat{\mathbf{r}} &= l_n \hat{\mathbf{n}} - l_p \hat{\mathbf{p}} \\ &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} - l_p \hat{\mathbf{p}} \\ &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} - (\hat{\mathbf{L}} - (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}}) \\ &= (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} - \hat{\mathbf{L}} + (\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} \\ &= 2(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}) \hat{\mathbf{n}} - \hat{\mathbf{L}} \end{aligned}$$

Computing the view vector involves having access to the camera location, so we can compute the normalized vector from the current sample location to the camera center. In an earlier section, camera (or "eye") space was mentioned as a common space in which we could compute our lighting. If we assume that the surface sample location is in camera space, this simplifies the process, because the center of the camera is the origin of view space. Thus, the view vector is then the origin minus the surface sample location; that is, the zero vector minus the sample location. Thus, in camera space, the view vector is simply the negative of the sample position treated as a vector and normalized.

The specular term itself is designed specifically to create an intensity distribution that reaches its maximum when the view vector $\hat{\mathbf{v}}$ is equal to $\hat{\mathbf{r}}$; that is, when the viewer is looking directly at the reflection of the light vector. The intensity distribution falls off toward zero rapidly as the angle between the two vectors increases, with a "shininess" control that adjusts how rapidly the intensity attenuates. The term is based on the following formula:

$$(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^{m_{shine}} = (\cos\theta)^{m_{shine}}$$

where θ is the angle between $\hat{\mathbf{r}}$ and $\hat{\mathbf{v}}$. The shininess factor m_{shine} controls the size of the highlight; a smaller value of m_{shine} leads to a larger, more diffuse

highlight, which makes the surface appear more dull and matte, whereas a larger value of m_{shine} leads to a smaller, more intense highlight, which makes the surface appear shiny. This shininess factor is considered a property of the surface material and represents how smooth the surface appears. Generally, the complete specular term includes a specular color defined on the material (M_S), which allows the highlights to be tinted a given color. The specular light color is often set to the diffuse color of the light, since a colored light generally creates a colored highlight. In practice, however, the specular color of the material is more flexible. Plastic and clear-coated surfaces (such as those covered with clear varnish), whatever their diffuse color, tend to have white highlights, while metallic surfaces tend to have tinted highlights. For a more detailed discussion of this and several other (more advanced) specular reflection methods, see Chapter 16 of Foley et al. [38].

A visual example of a sphere lit from a single light source providing only specular light is shown in Figure 8.11. The complete specular lighting term is

$$C_S = \begin{cases} i_L max(0, (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}}))^{m_{shine}} L_S M_S, & \text{if } \hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Note that, as with the diffuse term, a self-shadowing conditional is applied, ($\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0$). However, unlike the diffuse case, we must make this term explicit, as the specular term is not directly dependent upon $\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}$. Simply clamping the specular term to be greater than 0 could allow objects whose normals point away from the light to generate highlights, which is not correct. In other words, it is possible for $\hat{\mathbf{r}} \cdot \hat{\mathbf{v}} > 0$, even if $\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} < 0$.

In our pipeline, both materials and lights have specular components but only materials have specular exponents, as the specular exponent represents the shininess of a particular surface. We will store the specular color of an object's material in the 3-vector shader uniform value `materialSpecularColor`. The specular exponent material property is the scalar shader uniform `materialSpecularExp`. As previously noted for ambient and diffuse lighting, we will compute the specular component of a light by multiplying a scalar ambient light factor, `lightAmbDiffSpec.z`, times the light color, giving (`lightColor * lightAmbDiffSpec.z`).

The shader code to compute the specular component is as follows:

```
// GLSL Code
uniform vec3 materialSpecularColor;
uniform float materialSpecularExp;
uniform vec3 lightAmbDiffSpec;
uniform vec3 lightColor;

vec3 computeSpecularComponent(in vec3 surfaceNormal,
```

```
                                in vec4 surfacePosition,
                                in lightSampleValues light)
{
   vec3 viewVector = normalize(-surfacePosition.xyz);

   vec3 reflectionVector
        = 2.0 * dot(light.L, surfaceNormal) * surfaceNormal
          - light.L;

   return (dot(surfaceNormal, light.L) <= 0.0)
           ? vec3(0.0,0.0,0.0)
           : (light.iL * (lightColor * lightAmbDiffSpec.z)
              * materialSpecularColor
           * pow(max(0.0, dot(reflectionVector, viewVector)),
                    materialSpecularExp));
}
```

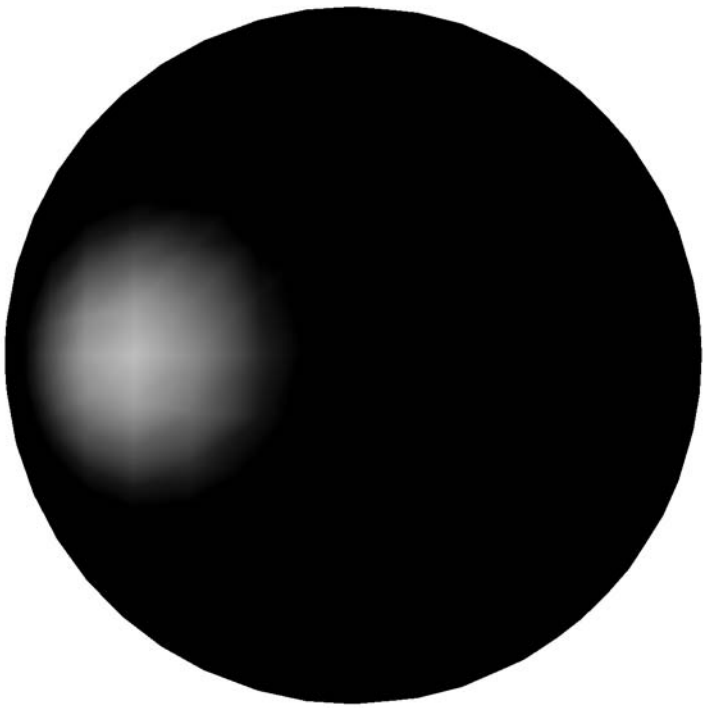


FIGURE 8.11 Sphere lit by specular light.

Infinite Viewer Approximation

One of the primary reasons that the specular term is the most expensive component of lighting is the fact that a normalized view and reflection vector must be computed for each sample, requiring at least one normalization per sample, per light. However, there is another method of approximating specular reflection that can avoid this expense in common cases. This method is based on a slightly different approximation to the specular highlight geometry, along with an assumption that the viewer is "at infinity" (at least for the purposes of specular lighting).

Rather than computing $\hat{\mathbf{r}}$ directly, the OpenGL method uses what is known as a *halfway vector*. The halfway vector is the vector that is the normalized sum of $\hat{\mathbf{L}}$ and $\hat{\mathbf{v}}$:

$$\hat{\mathbf{h}} = \frac{\hat{\mathbf{L}} + \hat{\mathbf{v}}}{|\hat{\mathbf{L}} + \hat{\mathbf{v}}|}$$

The resulting vector bisects the angle between $\hat{\mathbf{L}}$ and $\hat{\mathbf{v}}$. This halfway vector is equivalent to the surface normal $\hat{\mathbf{n}}$ that would generate $\hat{\mathbf{r}}$ such that $\hat{\mathbf{r}} = \hat{\mathbf{v}}$. In other words, given fixed light and view directions, $\hat{\mathbf{h}}$ is the surface normal that would produce the maximum specular intensity. So, the highlight is brightest when $\hat{\mathbf{n}} = \hat{\mathbf{h}}$. Figure 8.12 is a visual representation of the configuration, including the surface orientation of maximum specular reflection. The resulting (modified) specular term is

$$C_S = \begin{cases} i_L max(0, (\hat{\mathbf{h}} \cdot \hat{\mathbf{n}}))^{m_{shine}} L_S M_S, & \text{if } \hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

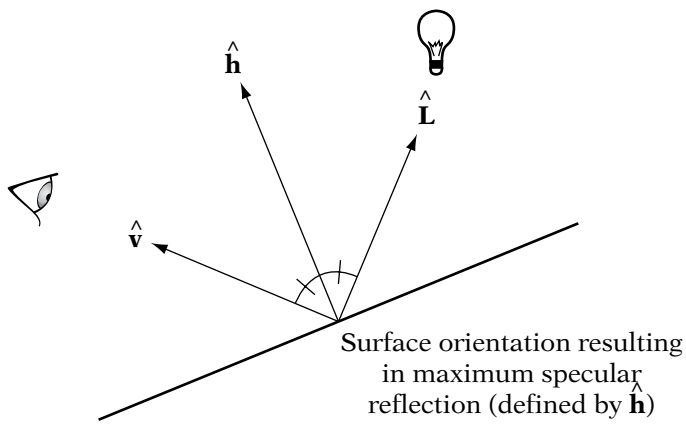


Figure 8.12 The specular halfway vector.

By itself, this new method of computing the specular highlight would not appear to be any better than the reflection vector system. However, if we assume that the viewer is at infinity, then we can use a constant view vector for all vertices, generally the camera's view direction. This is analogous to the difference between a point light and a directional (infinite) light. Thanks to the fact that the halfway vector is based only on the view vector and the light vector, the infinite viewer assumption can reap great benefits when used with directional lights. Note that in this case, both $\hat{\mathbf{L}}$ and $\hat{\mathbf{v}}$ are constant across all samples, meaning that the halfway vector $\hat{\mathbf{h}}$ is also constant. Used together, these facts mean that specular lighting can be computed very quickly if directional lights are used exclusively and the infinite viewer assumption is enabled. The halfway vector then can be computed once per object and passed down as a shader uniform, as follows:

```
// GLSL Code
uniform vec3 materialSpecularColor;
uniform float materialSpecularExp;
uniform vec3 lightAmbDiffSpec;
uniform vec3 lightColor;
uniform vec3 lightHalfway;

vec3 computeSpecularComponent(in vec3 surfaceNormal,
                              in lightSampleValues light)
{
   return (dot(surfaceNormal, light.L) <= 0.0)
           ? vec3(0.0,0.0,0.0)
           : (light.iL * (lightColor * lightAmbDiffSpec.z)
              * materialSpecularColor
           * pow(max(0.0, dot(surfaceNormal, lightHalfway)),
                  materialSpecularExp));
}
```

8.7 Combined Lighting Equation

Having covered materials, lighting components, and light sources, we now have enough information to evaluate our full lighting model for a given light at a given point.

$$\begin{aligned} C_V &= \text{Emissive} \\ &\quad + (\text{Ambient} + \text{Diffuse} + \text{Specular}) \\ &= M_E + (C_A + C_D + C_S) \\ A_V &= M_{Alpha} \end{aligned} \tag{8.1}$$

where the results are

1. C_V, the computed, lit RGB color of the sample.
2. A_V, the alpha component of the RGBA color of the sample.

The intermediate, per-light values used to compute the results are

3. C_A, the ambient light term, which is equal to

$$C_A = i_L M_A L_A$$

4. C_D, the diffuse light term, which is equal to

$$C_D = i_L M_D L_D (max(0, \hat{\mathbf{L}} \cdot \hat{\mathbf{n}}))$$

5. C_S, the specular light term, which is equal to

$$C_S = i_L M_S L_S \begin{cases} max(0, (\hat{\mathbf{h}} \cdot \hat{\mathbf{n}}))^{m_{shine}}, & \text{if } \hat{\mathbf{L}} \cdot \hat{\mathbf{n}} > 0 \\ 0, & \text{otherwise} \end{cases}$$

The shader code to compute this, based upon the shader functions already defined previously, is as follows:

```
// GLSL Code

vec3 computeLitColor(in lightSampleValues light,
                     in vec4 surfacePosition,
                     in vec3 surfaceNormal)
{
   return computeAmbientComponent(light)
         + computeDiffuseComponent(surfaceNormal, light)
         + computeSpecularComponent(surfaceNormal,
                                    surfacePositon,
                                    light);
}

// ...

uniform vec3 materialEmissiveColor;
uniform vec4 materialDiffuseColor;

   vec4 finalColor;
   finalColor.rgb = materialEmissiveColor
                  + computeLitColor(light, pos, normal);
   finalColor.a = materialDiffuseColor.a;
```

SOURCE CODE DEMO MultipleLights

For a visual example of all of these components combined, see the lit sphere in Figure 8.13.

Most interesting scenes will contain more than a single light source. Thus, the lighting model and the code must take this into account. When lighting a given point, the contributions from each component of each active light L are summed to form the final lighting equation, which is detailed as follows:

$$\begin{aligned} C_V &= \text{Emissive} + \text{Ambient} \\ &\quad + \sum_{L}^{lights} (\text{Per-light Ambient} + \text{Per-light Diffuse} + \text{Per-light Specular}) \\ &= M_E + \sum_{L}^{lights} (C_A + C_D + C_S) \\ A_V &= M_{Alpha} \end{aligned} \tag{8.2}$$

The combined lighting equation 8.2 brings together all of the properties discussed in the previous sections. In order to implement this equation in

FIGURE 8.13 Sphere lit by a combination of ambient, diffuse, and specular lighting.

shader code, we need to compute i_L and $\hat{\mathbf{L}}$ per active light. The shader code for computing these values required source data for each light. In addition, the type of data required differed by light type. The former issue can be solved by passing arrays of uniforms for each value required by a light type. The elements of the arrays represent the values for each light, indexed by a loop variable. For example, if we assume that all of our lights are directional, the code to compute the lighting for up to eight lights might be as follows:

```
// GLSL Code
uniform vec3 materialEmissiveColor;
uniform vec3 materialAmbientColor;
uniform vec4 materialDiffuseColor;
uniform vec3 materialSpecularColor;
uniform float materialSpecularExp;
uniform int dirLightCount;
uniform vec4 dirLightPosition[8];
uniform float dirLightIntensity[8];
uniform vec3 lightAmbDiffSpec[8];
uniform vec3 lightColor[8];

lightSampleValues computeDirLightValues(in int i)
{
   lightSampleValues values;
   values.L = dirLightPosition[i];
   values.iL = dirLightIntensity[i];
   return values;
}

vec3 computeAmbientComponent(in lightSampleValues light,
                             in int i)
{
   return light.iL * (lightColor[i] * lightAmbDiffSpec[i].x)
                   * materialAmbientColor;
}

vec3 computeDiffuseComponent(in vec3 surfaceNormal,
                             in lightSampleValues light,
                             in int i)
{
   return light.iL * (lightColor[i] * lightAmbDiffSpec[i].y)
                   * materialDiffuseColor.rgb
                   * max(0.0, dot(surfaceNormal, light.L));
}

vec3 computeSpecularComponent(in vec3 surfaceNormal,
```

```
                                in vec4 surfacePositon,
                                in lightSampleValues light,
                                in int i)
{
   vec3 viewVector = normalize(-surfacePosition.xyz);
   vec3 reflectionVector
        = 2.0 * dot(light.L, surfaceNormal) * surfaceNormal
          - light.L;
   return (dot(surfaceNormal, light.L) <= 0.0)
           ? vec3(0.0,0.0,0.0)
           : (light.iL * (lightColor[i] * lightAmbDiffSpec[i].z)
              * materialSpecularColor
           * pow(max(0.0, dot(reflectionVector, viewVector)),
                   materialSpecularExp));
}

vec3 computeLitColor(in lightSampleValues light,
                     in vec4 surfacePosition,
                     in vec3 surfaceNormal, in int i)
{
   return computeAmbientComponent(light, i)
         + computeDiffuseComponent(surfaceNormal, light, i)
         + computeSpecularComponent(surfaceNormal,
                                    surfacePositon,
                                    light, i);
}


{
   int i;
   vec4 finalColor;
   finalColor.rgb = materialEmissiveColor;
   finalColor.a = materialDiffuseColor.a;
   for (i = 0; i < dirLightCount; i++)
   {
      lightSampleValues light = computeDirLightValues(i);
      finalColor.rgb + =
                  computeLitColor(light, i, pos, normal);
   }
}
```

The code becomes even more complex when we must consider different types of light sources. One approach to this is to use independent arrays for each type of light and iterate over each array independently. The complexity of

these approaches and the number of uniforms that must be sent to the shader can be prohibitive for some systems. As a result, it is common for rendering engines to either generate specific shaders for the lighting cases they know they need, or else generate custom shader source code in the engine itself, compiling these shaders at runtime as they are required.

Clearly, many different values and components must come together to light even a single sample. This fact can make lighting complicated and difficult to use at first. A completely black rendered image or a flat-colored resulting object can be the result of many possible errors. However, an understanding of the lighting pipeline can make it much easier to determine which features to disable or change in order to debug lighting issues.

8.8 Lighting and Shading

Thus far, our lighting discussion has focused on computing color at a generic point on a surface, given a location, surface normal, view vector, and surface material. We have specifically avoided specifying whether these code snippets in our shader code examples are to be vertex or fragment shaders. Another aspect of lighting that is just as important as the basic lighting equation is the question of when and how to evaluate that equation to completely light a surface. Furthermore, if we do not choose to evaluate the full lighting equation at every sample point on the surface, how do we interpolate or reuse the explicitly lit sample points to compute reasonable colors for these other samples.

Ultimately, a triangle in view is drawn to the screen by coloring the screen pixels covered by that triangle (as will be discussed in more detail in Chapter 9). Any lighting system must be teamed with a shading method that can quickly compute colors for each and every pixel covered by the triangle. These shading methods determine when to invoke the shader to compute the lighting and when to simply reuse or interpolate already computed lighting results from other samples. In most cases, this is a performance versus visual accuracy trade-off, since it is normally more expensive computationally to evaluate the shader than it is to reuse or interpolate already computed lighting results.

The sheer number of pixels that must be drawn per frame requires that low- to mid-end graphics systems forego computing more expensive lighting equations for each pixel in favor of another method. For example, a sphere that covers 50 percent of a mid-sized $1{,}280 \times 1{,}024$ pixel screen will require the shading system to compute colors for over a half million pixels, regardless of the tessellation. Next, we will discuss some of the more popular methods. Some of these methods will be familiar, as they are simply the shading methods discussed in Chapter 7, using results of the lighting equation as source colors.

8.8.1 Flat-Shaded Lighting

Historically, the simplest shading method applied to lighting was per-triangle, flat shading. This method involved evaluating the lighting equation once per triangle and using the resulting color as the constant triangle color. This color is assigned to every pixel covered by the triangle. In older, fixed-function systems, this was the highest-performance lighting/shading combination, owing to two facts: the more expensive lighting equation needed only to be evaluated once per triangle, and a single color could be used for all pixels in the triangle. Figure 8.14 shows an example of a sphere lit and shaded using per-triangle lighting and flat shading.

To evaluate the lighting equation for a triangle, we need a sample location and surface normal. The surface normal used is generally the triangle face normal (discussed in Chapter 2), as it accurately represents the plane of the triangle. However, the issue of sample position is more problematic. No single point can accurately represent the lighting across an entire triangle (except in special cases); for example, in the presence of a point light, different points on the triangle should be attenuated differently, according to their distance

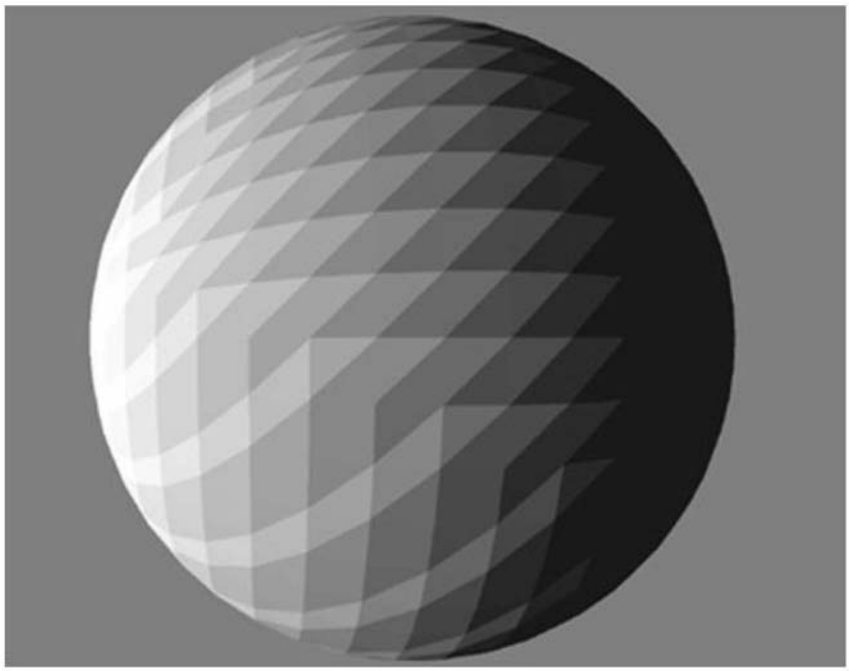


Figure 8.14 Sphere lit and shaded by per-triangle lighting and flat shading.

from the light. While the centroid of the triangle is a reasonable choice, the fact that it must be computed specifically for lighting makes it less desirable. For reasons of efficiency (and often to match with the graphics system), the most common sample point for flat shading is one of the triangle vertices, as the vertices already exist in the desired space. This can lead to artifacts, since a triangle's vertices are (by definition) at the edge of the area of the triangle. Flat-shaded lighting does not match quite as well with modern programmable shading pipelines, and the simplicity of the resulting lighting has meant that it is of somewhat limited interest in modern rendering systems.

8.8.2 Per-Vertex Lighting

Flat-shaded lighting suffers from the basic flaws and limitations of flat shading itself; the faceted appearance of the resulting geometry tends to highlight rather than hide the piecewise triangular approximation. In the presence of specular lighting, the tessellation is even more pronounced, causing entire triangles to be lit with bright highlights. With moving lights or geometry, this can cause gemstonelike "flashing" of the facets. For smooth surfaces such as the sphere in Figure 8.14 this faceting is often unacceptable.

The next logical step is to use per-vertex lighting with Gouraud interpolation of the resulting color values. The lighting equation is evaluated in the vertex shader, and the resulting color is passed as an interpolated varying color to the simple fragment shader. The fragment shader can be extremely simple, doing nothing more than assigning the interpolated varying color as the final fragment color.

Generating a single lit color that is shared by all colocated vertices leads to smooth lighting across surface boundaries. Even if colocated vertices are not shared (i.e., each triangle has its own copy of its three vertices), simply setting the normals to be the same in all copies of a vertex will cause all copies to be lit the same way. Figure 8.15 shows an example of a sphere lit and shaded using per-vertex lighting and Gouraud shading.

Per-vertex lighting only requires evaluating the lighting equation once per vertex. In the presence of well-optimized vertex sharing (where there are more triangles than vertices), per-vertex lighting can actually require *fewer* lighting equation evaluations than does true per-triangle flat shading. The interpolation method used to compute the per-fragment varying values (Gouraud) is more expensive computationally than the trivial one used for flat shading, since it must interpolate between the three vertex colors on a per-pixel basis. However, modern shading hardware is heavily tuned for this form of varying value interpolation, so the resulting performance of per-vertex lighting is generally close to peak.

Gouraud-shaded lighting is a vertex-centric method — the surface positions and normals are used only at the vertices, with the triangles serving

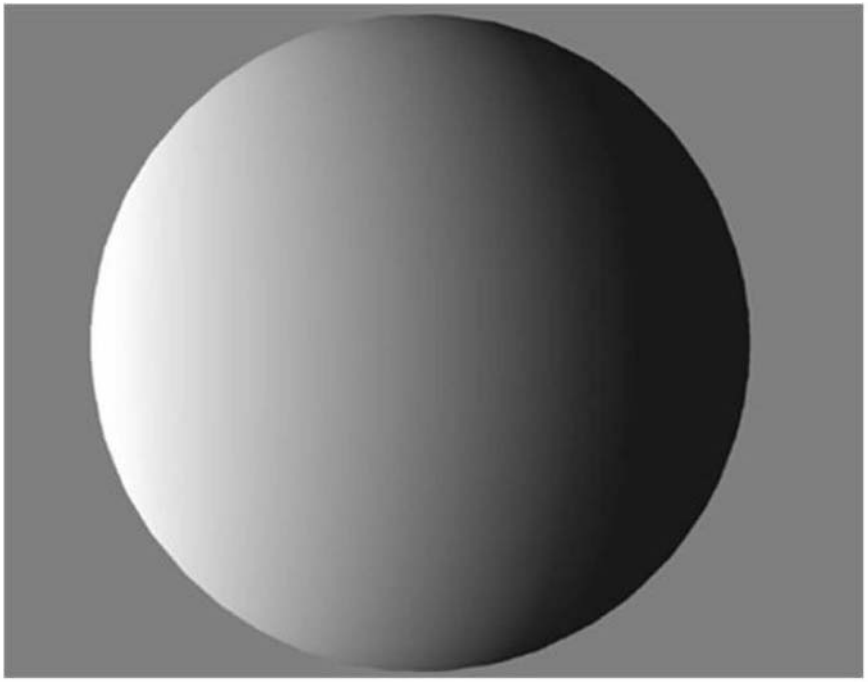


Figure 8.15 Sphere lit and shaded by per-vertex lighting and Gouraud shading.

only as areas for interpolation. This shift to vertices as localized surface representations lends focus to the fact that we will need smooth surface normals at each vertex. The next section will discuss several methods for generating these vertex normals.

Generating Vertex Normals

In order to generate smooth lighting that represents a surface at each vertex, we need to generate a single normal that represents the surface at each vertex, not at each triangle. There are several common methods used to generate these per-vertex surface normals at content creation time or at load time, depending upon the source of the geometry data.

When possible, the best way to generate smooth normals during the creation of a tessellation is to use analytically computed normals based on the surface being approximated by triangles. For example, if the set of triangles represent a sphere centered at the origin, then for any vertex at location P_V, the surface normal is simply

$$\hat{\mathbf{n}} = \frac{P_V - 0}{|P_V - 0|}$$

This is the vertex position, treated as a vector (thus the subtraction of the zero point) and normalized. Analytical normals can create very realistic impressions of the original surface, as the surface normals are pivotal to the overall lighting impression. Examples of surfaces for which analytical normals are available include implicit surfaces and parametric surface representations, which generally include analytically defined normal vectors at every point in their domain.

In the more common case, the mesh of triangles exists by itself, with no available method of computing exact surface normals for the surface being approximated. In this case, the normals must be generated from the triangles themselves. While this is unlikely to produce optimal results in all cases, simple methods can generate normals that tend to create the impression of a smooth surface and remove the appearance of faceting.

One of the most popular algorithms for generating normals from triangles takes the mean of all of the face normals for the triangles that use the given vertex. Figure 8.16 demonstrates a two-dimensional (2D) example of averaging triangle normal vectors. The algorithm may be pseudo-coded as follows:

```
for each vertex V
{
    vector V.N = (0,0,0);
    for each triangle T that uses V
    {
      vector F = TriangleNormal(T);
      V.N += F;
    }     V.N.Normalize();
}
```

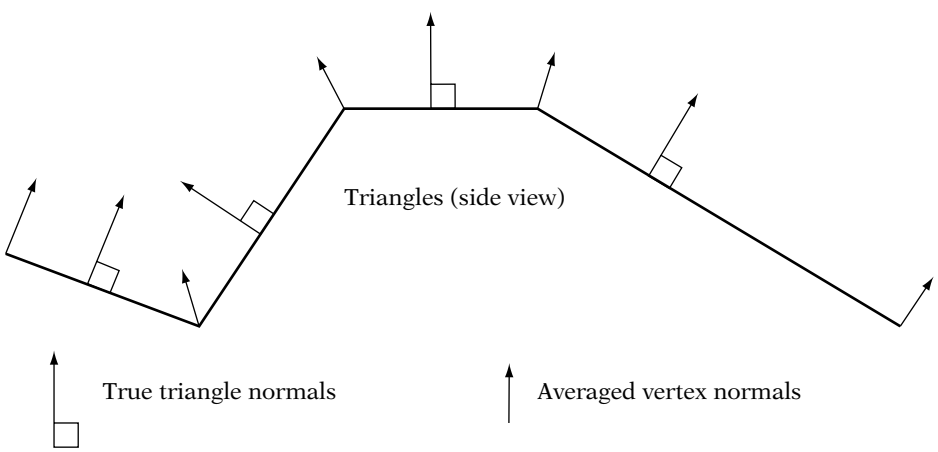


Figure 8.16 Averaging triangle normal vectors.

Basically, the algorithm sums the normals of all of the faces that are incident upon the current vertex and then renormalizes the resulting summed vector. Since this algorithm is (in a sense) a mean-based algorithm, it can be affected by tessellation. Triangles are not weighted by area or other such factors, meaning that the face normal of each triangle incident upon the vertex has an equal "vote" in the makeup of the final vertex normal. While the method is far from perfect, any vertex normal generated from triangles will by its nature be an approximation. In most cases, the averaging algorithm generates convincing normals. Note that in cases where there is no fast (i.e., constant-time) method of retrieving the set of triangles that use a given vertex (e.g., if only the OpenGL/Direct3D-style index lists are available), the algorithm may be turned "inside out" as follows:

```
for each vertex V
{
    V.N = (0,0,0);
}
for each triangle T
{
    // V1, V2, V3 are the vertices used by the triangle
    vector F = TriangleNormal(T);
    V1.N += F;
    V2.N += F;
    V3.N += F;
}
for each vertex V
{
    V.N.Normalize();
}
```

Basically, this version of the algorithm uses the vertex normals as "accumulators," looping over the triangles, adding each triangle's face normal to the vertex normals of the three vertices in that triangle. Finally, having accumulated the input from all triangles, the algorithm goes back and normalizes each final vertex normal. Both algorithms will result in the same vertex normals, but each works well with different vertex/triangle data structure organizations.

Sharp Edges

As with Gouraud shading based on fixed colors, Gouraud-shaded lighting with vertices shared between triangles generates smooth triangle boundaries by default. In order to represent a sharp edge, vertices along a physical crease in the geometry must be duplicated so that the vertices can represent the surface

normals on either side of the crease. By having different surface normals in copies of colocated vertices, the triangles on either side of an edge can be lit according to the correct local surface orientation. For example, at each vertex of a cube, there will be three vertices, each one with a normal of a different face orientation, as we see in Figure 8.17.

8.8.3 Per-Fragment Lighting

Source Code
DEMO
PerFragmentLighting

There are significant limitations to per-vertex lighting. Specifically, the fact that the lighting equation is evaluated only at the vertices can lead to artifacts. Even a cursory evaluation of the lighting equation shows that it is highly nonlinear. However, Gouraud shading interpolates linearly across polygons. Any nonlinearities in the lighting across the interior of the triangle will be lost completely. These artifacts are not as noticeable with diffuse and ambient lighting as they are with specular lighting, because diffuse and ambient lighting are closer to linear functions than is specular lighting (owing at least partially to

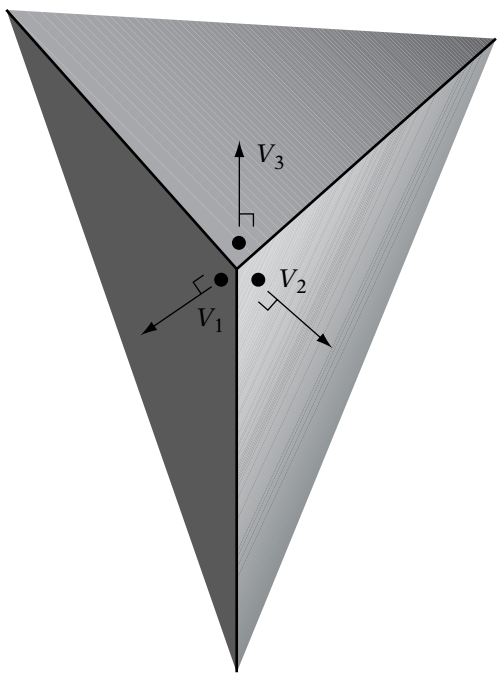


Figure 8.17 One corner of a faceted cube.

the nonlinearity of the specular exponent term and to the rapid changes in the specular halfway vector $\hat{\mathbf{h}}$ with changes in viewer location).

For example, let us examine the specular lighting term for the surface shown in Figure 8.18. We draw the 2D case, in which the triangle is represented by a line segment. In this situation, the vertex normals all point outward from the center of the triangle, meaning that the triangle is representing a somewhat curved (domed) surface. The point light source and the viewer are located at the same position in space, meaning that the view vector $\hat{\mathbf{v}}$, the light vector $\hat{\mathbf{L}}$, and the resulting halfway vector $\hat{\mathbf{h}}$ will all be equal for all points in space. The light and viewer are directly above the center of the triangle. Because of this, the specular components computed at the two vertices will be quite dark (note the specular halfway vectors shown in Figure 8.18 are almost perpendicular to the normals at the vertices). Linearly interpolating between these two dark specular vertex colors will result in a polygon that is relatively dark.

However, if we look at the geometry that is being approximated by these normals (a domed surface as in Figure 8.18), we can see that in this configuration the interpolated normal at the center of the triangle would point straight up at the viewer and light. If we were to evaluate the lighting equation at a

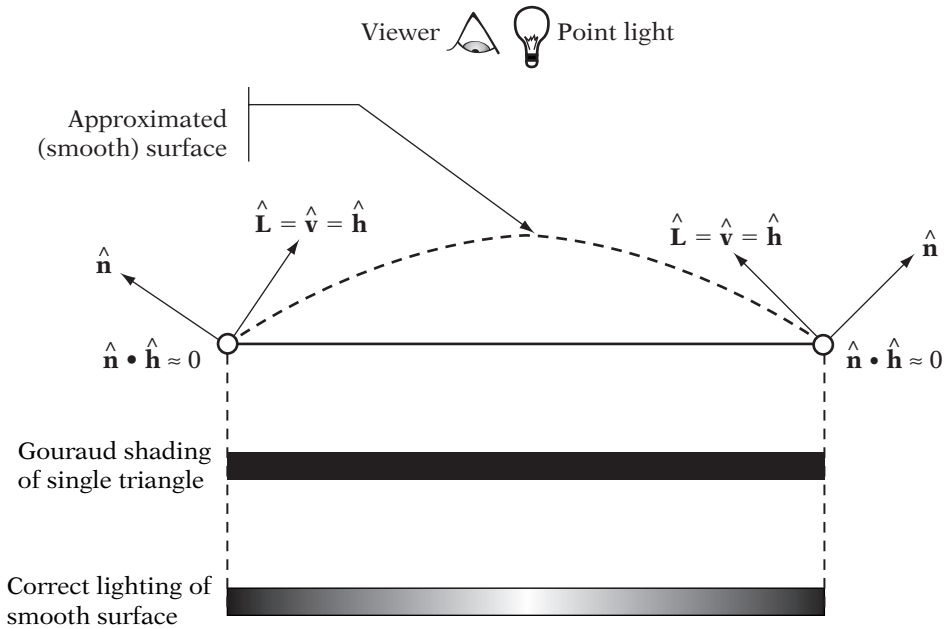


FIGURE 8.18 Gouraud shading can miss specular highlights.

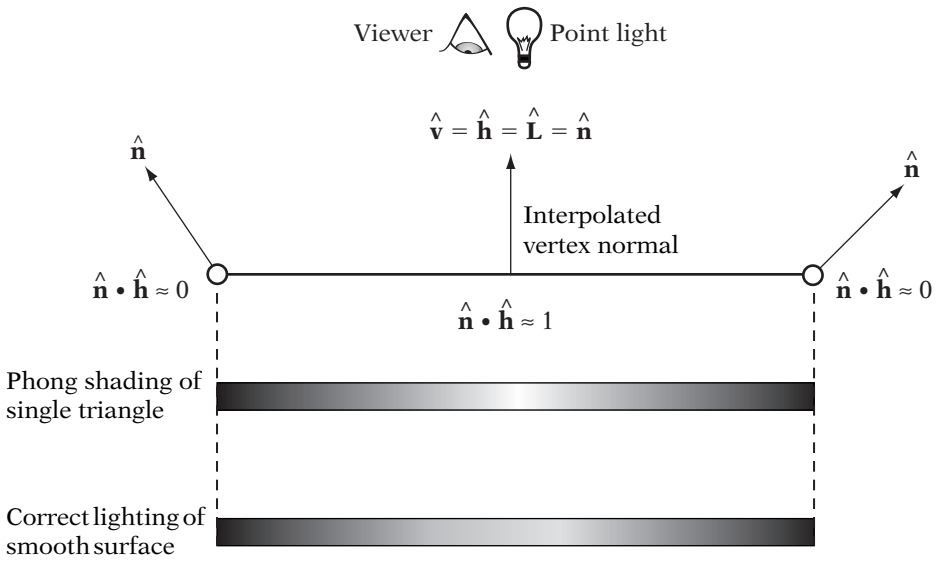


Figure 8.19 Phong shading of the same configuration.

point near the center of the triangle in this case, we would find an extremely bright specular highlight there. The specular lighting across the surface of this triangle is highly nonlinear, and the maximum is internal to the triangle. Even more problematic is the case in which the surface is moving over time. In rendered images where the highlight happens to line up with a vertex, there will be a bright, linearly interpolated highlight at the vertex. However, as the surface moves so that the highlight falls between vertices, the highlight will disappear completely. This is a very fundamental problem with approximating a complex function with a piecewise linear representation. The accuracy of the result is dependent upon the number of linear segments used to approximate the function. In our case, this is equivalent to the density of the tessellation.

If we want to increase the accuracy of lighting on a general vertex-lit surface, we must subdivide the surface to increase the density of vertices (and thus lighting samples). However, this is an expensive process, and we may not know a priori which sections of the surface will require significant tessellation. Dependent upon the particular view at runtime, almost any tessellation may be either overly dense or too coarse. In order to create a more general, high-quality lighting method, we must find another way around this problem.

So far, the methods we have discussed for lighting have all evaluated the lighting equation once per basic geometric object, such as per vertex or per triangle. *Phong shading* (named after its inventor, Bui Phong Tuong [93])

works by evaluating the lighting equation once for each fragment covered by the triangle. The difference between Gouraud and Phong shading may be seen in Figures 8.18 and 8.19. For each sample across the surface of a triangle, the vertex normals, positions, reflection, and view vectors are interpolated, and the interpolated values are used to evaluate the lighting equation. However, since triangles tend to cover more than 1–3 pixels, such a lighting method will result in far more lighting computations per triangle than do per-triangle or per-vertex methods.

Per-fragment lighting changes the balance of the work to be done in the vertex and fragment shaders. Instead of computing the lighting in the vertex shader, per-pixel lighting uses the vertex shader only to set up the source values (surface position, surface normal, view vector) and pass them down as varying values to the fragment shader. As always, the varying values are interpolated using Gouraud interpolation and passed to each invocation of the fragment shaders. These interpolated values now represent smoothly interpolated position and normal vectors for the surface being represented. It is these values that are used as sources to the lighting computations, evaluated in the fragment shader.

There are several issues that make Phong shading more computationally expensive than per-vertex lighting. The first of these is the actual normal vector interpolation, since basic barycentric interpolation of the three vertex normals will almost *never* result in a normalized vector. As a result, the interpolated normal vector will have to be renormalized *per fragment*, which is much more frequently than per vertex.

Furthermore, the full lighting equation must be evaluated per sample once the interpolated normal is computed and renormalized. Not only is this operation expensive, it is not a fixed amount of computation. As we saw above, in a general engine, the complexity of the lighting equation is dependent on the number of lights and numerous graphics engine settings. This resulted in Phong shading being rather unpopular in game-centric consumer 3D hardware prior to the advent of pixel and vertex shaders.

An example of a vertex shader passing down the required camera-space positions and normals is as follows:

```
// GLSL
varying vec4 lightingPosition;
varying vec3 lightingNormal;

void main()
{
   // The position and normal for lighting
   // must be in camera space, not homogeneous space
   lightingPosition = gl_ModelViewMatrix * gl_Vertex;
```

```
    lightingNormal = gl_NormalMatrix * gl_Normal;

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

An example of a fragment shader implementing lighting by a single directional light is as follows:

```
// GLSL Code
uniform vec3 materialEmissiveColor;
uniform vec3 materialAmbientColor;
uniform vec4 materialDiffuseColor;
uniform vec3 materialSpecularColor;
uniform vec4 dirLightPosition;
uniform float dirLightIntensity;
uniform vec3 lightAmbDiffSpec;
uniform vec3 lightColor;

varying vec4 lightingPosition;
varying vec3 lightingNormal;

{
    vec4 finalColor;
    finalColor.rgb = materialEmissiveColor;
    finalColor.a = materialDiffuseColor.a;

    vec3 interpolatedNormal = normalize(lightingNormal);

    lightSampleValues light = computeDirLightValues();
    finalColor.rgb += computeLitColor(light,
       lightingPosition, interpolatedNormal);
    }
}
```

8.9 Textures and Lighting

Of the methods we have discussed for coloring geometry, the two most powerful are texturing and dynamic lighting. However, they each have drawbacks when used by themselves. Texturing is normally a static method and looks flat and painted when used by itself in a dynamic scene. Lighting can generate very dynamic effects, but when limited to per-face-level or per-vertex-level features, can lead to limited detail. It is only natural that graphics systems would want to use the results of both techniques together on a single surface.

8.9.1 Basic Modulation

The simplest methods of merging these two techniques involves simply viewing each of the two methods as generating a color per sample and merging them. With texturing, this is done directly via texture sampling; with lighting, it is done by evaluating the lighting equation. These two colors must be combined in a way that makes visual sense. The most common way of combining textures and lighting results is via multiplication, or "modulation." In modulate lighting/texture combination, the texture color at the given sample C_T and the final lit and/or interpolated vertex color are combined by per-component multiplication.

The visual effect here is that the per-vertex lit colors darken the texture (or vice versa). As a result, texture images designed to be used with modulate mode texture combination are normally painted as if they were fully lit. The vertex colors, representing the lighting in the scene, darken these fully lit textures to make them look more realistic in the given environment. As Figure 8.20 demonstrates, the result of modulation can be very convincing, even though the lighting is rather simple and the textures are static paintings. In the presence of moving or otherwise animated lights, the result can be even more immersive, as the human perceptual system is very reliant upon lighting cues in the real world. This is, in a sense, using the texture as a factor in all of the surface material colors.

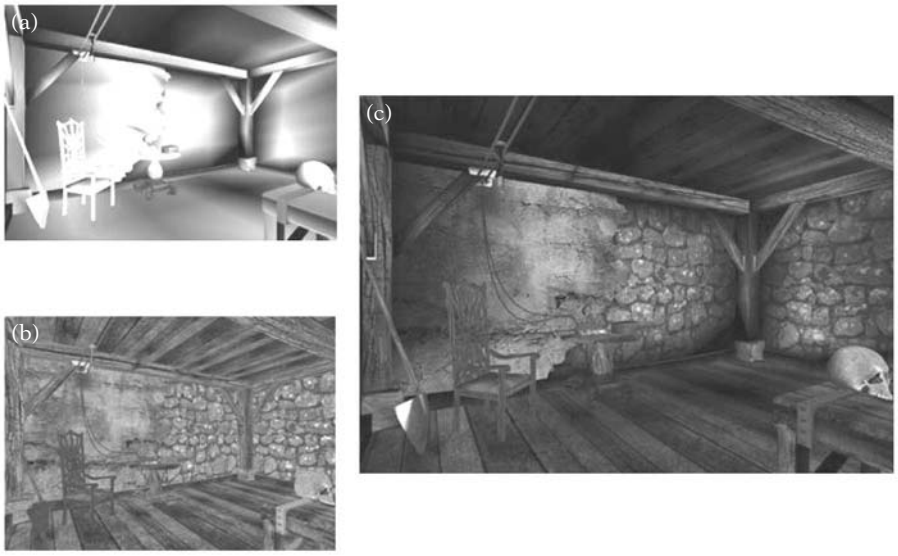


Figure 8.20 Textures and lighting combined via modulation: (a) scene with pure vertex lighting, (b) scene with pure texturing, and (c) same scene with lighting and texturing combined.

Assuming for the moment that the lit color is computed and stored in `litColor`, either by computation in the fragment shader or passed down as a varying component, a simple textured, lit fragment shader would be as follows:

```
// GLSL - fragment shader
uniform sampler2D texture;
varying vec2 texCoords;

void main()
{
   // lit color is in vec4 litColor
   vec4 litColor;

   // ...

   // Sample the texture represented by "texture"
   // at the location "texCoords"
   gl_FragColor = litColor * texture2D (texture, texCoords);
}
```

Until the advent of programmable shaders, modulation was the most popular and often the only method of combining lighting and textures.

8.9.2 Specular Lighting and Textures

If the full lighting equation 8.1 is combined with the texture via multiplication, then lighting can only *darken* the texture, since lit vertex colors are clamped to the range [0, 1]. While this looks correct for diffuse or matte objects, for shiny objects with bright specular highlights, it can look very dull. It is often useful to have the specular highlights "wash out" the texture. We cannot simply add the full set of lighting because the texture will almost always wash out and can *never* get darker. To be able to see the full range of effects requires that the diffuse colors darken the texture while the specular components of color add highlights. This is only possible if we split the lighting components.

Because the specular term is added after the texture is multiplied, this mode (sometimes called *modulate with late add*) causes the diffuse terms to attenuate the texture color, while the specular terms wash out the result. The differences between the separate and combined specular modes can be very striking, as Figure 8.21 makes clear.

The shader code to compute this involves computing the emissive, ambient, and diffuse lighting components into one color (which we'll call

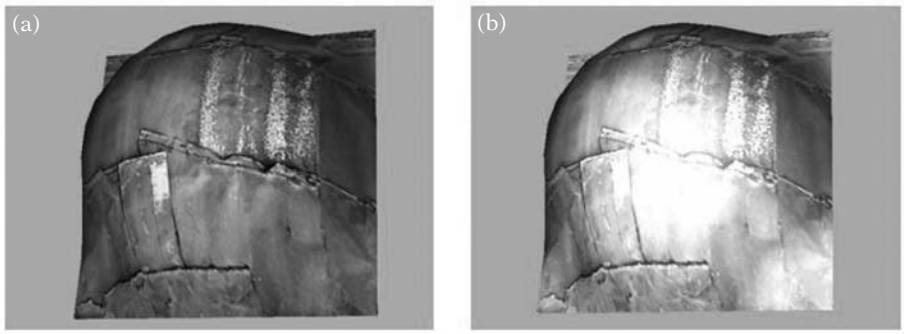


FIGURE 8.21 Combining textures and lighting: (a) specular vertex color added to diffuse vertex color, then modulated with the texture, and (b) diffuse vertex color modulated with the texture, then specular vertex color added.

`diffuseLighting`) and the specular component into another (which we'll call `specularLighting`). Having computed these independently, we merge them as follows:

```
// GLSL - fragment shader
uniform sampler2D texture;
varying vec2 texCoords;

void main()
{
   vec4 diffuseLighting;
   vec4 specularLighting;

   // ...

   // Sample the texture represented by "texture"
   // at the location "texCoords"
   gl_FragColor = diffuseLighting * texture2D (texture, texCoords)
                  + specularLighting;
}
```

In this case, the texture is providing a scaling factor for only the emissive, ambient, and diffuse material color. The effect is simple to add to an existing lighting and texture shader and can make for a much more dynamic result.

8.9.3 Textures as Materials

The next step in using textures and lighting together involves using multiple textures on a single surface. As shown in the previous section, a texture can be used to modulate one or more material colors on a surface. In fact, textures also can be used to *replace* one or more surface material components. Common surface material colors to be replaced with textures are:

- **Material diffuse color.** Often called a *diffuse map*, this is extremely similar to basic modulation as shown above. Frequently, the diffuse map is also applied as the ambient material color.
- **Material specular color.** This is frequently replaced with either an RGB texture (a specular map) or a single-channel grayscale texture, which is called a *gloss map*. The gloss map is a powerful technique: Wherever it is close to full brightness, the object appears glossy, because specular highlights can be seen at those points on the surface. Wherever it is close to black, the specular highlights do not appear. As a result, it can be used to mark shiny or worn sections of an object, independent of the color. Frequently, gloss maps are created in such a way that they are brightest on the edges and exposed areas, parts of a surface likely to be worn down by the elements and naturally polished.
- **Material emissive color.** Often called a *glow map*, this texture can be used to localize self-illumination of an object. These maps are frequently used to mark windows in nighttime views of vehicles or buildings, or taillights and running lights of vehicles.

Since multiple textures can be used in a single shader, any or all of these components can be easily replaced by individual textures. The uniform material color vectors simply become texture sampler uniforms. Many of these textures can reuse the same texture coordinates, as the mappings of each texture can be the same. Finally, optimizations are common. For example, it is common to use an RGBA texture in which the RGB components are used as the diffuse map, and the alpha component is used as a single-channel gloss map. The ease of painting these components into a single texture can assist the artists, and the reuse of an otherwise unused texture component can save graphics processing unit (GPU) resources. An example of a fragment shader using RGBA diffuse and gloss maps is shown below:

```
// GLSL - fragment shader
uniform sampler2D texture;
varying vec2 texCoords;
```

```
void main()
{
   vec4 diffuseLighting;
   vec4 specularLighting;

   // ...
   vec4 diffuseAndGlossMap = texture2D (texture, texCoords);

   // Sample the texture represented by "texture"
   // at the location "texCoords"
   gl_FragColor.rgb = diffuseLighting.rgb * diffuseAndGlossMap.rgb
                    + specularLighting.rgb * diffuseAndGlossMap.a;
}
```

8.10 Advanced Lighting

Programmable shaders make an almost endless array of lighting effects possible. We will discuss a few of these methods and mention several others, citing references for additional information. Like the methods mentioned in the previous sections, many of these methods involve using textures as sources to the lighting equation.

8.10.1 Normal Mapping

So far, we have shown how one or more textures can be used to replace material colors and/or intensities in the lighting equation. However, even more advanced techniques are based on the fact that textures can be used to store more general values than mere colors. The most popular of these techniques are *bump mapping* and *normal mapping*. As the names suggest, these methods simulate bumpy surfaces by storing the detailed "height offsets" (bump mapping) or normal vectors (normal mapping) for the surface in a texture. One of the basic limitations of dynamic lighting as discussed so far is that while we can evaluate lighting on a per-fragment basis, the source values describing the geometry of the surface are interpolated from per-vertex values — the position and normal. As a result, a rough or bumpy surface requires a very high density of vertices. We can simulate this effect at a finer level by adding bumpy, pre-lit colors to the diffuse map texture, but in the presence of moving or changing lighting conditions, the static nature of this trick is obvious and jarring. Bump mapping, the first of these techniques to be available, was actually present in some fixed-function rendering hardware in the late 1990s. However, bump mapping, since it represented a local height offset and generated surface normals implicitly by looking at the difference in height values

between neighboring texels, was limited to surfaces that looked embossed. Very sharp changes in surface orientation were difficult with bump mapping. For a discussion of these limitations, see Theodore [110].

In order to add more detail to the lighting at this fine level, we must be able to generate a surface normal per fragment that contains real information (not just interpolated information) for each fragment. By storing the normal in a texture, we can generate normals that change very rapidly per fragment and respond correctly to changing lighting configurations. The normal vectors across the surface are stored in the RGB components of the texture (either as signed fixed-point values or as floating-point values). The exact space in which the normals are stored in the texture differs from method to method. Conceptually, the simplest space is object space, in which normals are computed in the model or object space of the geometry being mapped and then stored as `(x,y,z)` in the R, G, and B components of the texture, respectively. Object-space normal maps can be sampled as a regular texture into a `vec3` and then used in the same way as one would use normals passed into a fragment shader as a varying value. An example of the effect that normal mapping can have on a simple object is shown in Figure 8.22

An example of an object-space normal map in use in a fragment shader is shown below. Note that since the sampled normal comes from the texture in object space, we must either do the transformation of the normal into camera space in the fragment shader, or else transform the light information into model space and light in model space. We pick the former (which is simpler to

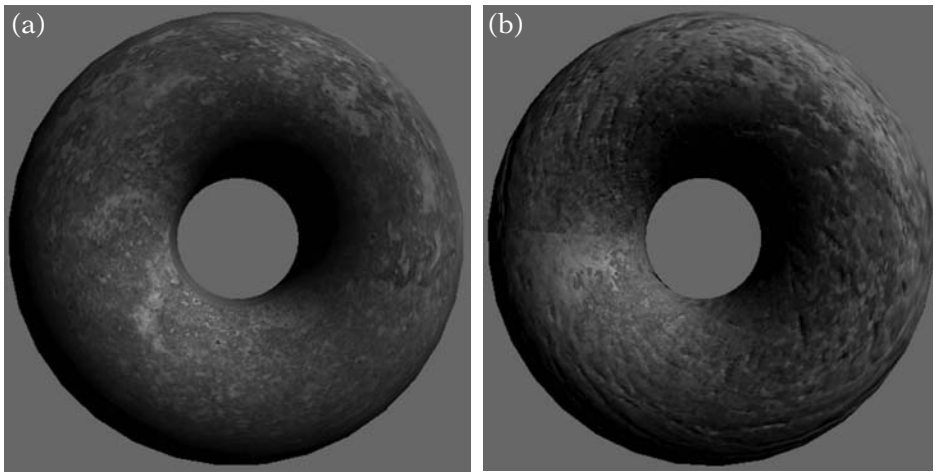


Figure 8.22 Normal mapping applied to simple objects: (a) low triangle—count object with diffuse map and per-vertex lighting, and (b) same object with diffuse map and per-fragment normal mapping.

understand, but more expensive computationally) and transform the normal into camera space.

```
// GLSL Code
uniform sampler2D normalMap; varying vec2 texCoords;
uniform vec3 materialEmissiveColor;
uniform vec3 materialAmbientColor;
uniform vec4 materialDiffuseColor;
uniform vec3 materialSpecularColor;
uniform vec4 dirLightPosition;
uniform float dirLightIntensity;
uniform vec3 lightAmbDiffSpec;
uniform vec3 lightColor;

varying vec4 lightingPosition;

{
   vec4 finalColor;
   finalColor.rgb = materialEmissiveColor;
   finalColor.a = materialDiffuseColor.a;

   // lookup the normal in the map, transform, and renormalize
   vec3 normal = texture2D (normalMap, texCoords);
   normal = normalize(gl_NormalMatrix * normal);

   lightSampleValues light = computeDirLightValues();
   finalColor.rgb += computeLitColor(light,
      lightingPosition, normal);
   }
}
```

There are advantages and disadvantages to both object-space normal mapping and the other common normal mapping method, known as *tangent space* normal maps (the latter is now the more popular of the two, but is conceptually a bit harder to understand). These differences are discussed in various articles on normal mapping [36, 92].

Generating Normal Maps

Normal maps are rarely painted by an artist. The complexity of these maps and the esoteric spaces in which they reside mean that most normal maps are generated automatically from the geometry itself via commercial or open-source tools. Some real-time 3D engines and middleware provide tools that

automatically convert a *very* high polygon–count object (millions of triangles) into a low polygon–count geometry object and a high-resolution normal map. Put together, the low-resolution geometry object and the high-resolution normal map can be used to efficiently render what appears to be a very convincing facsimile of the original object.

8.11 Reflective Objects

While specular lighting can provide the basic impression of a shiny object, large expanses of a reflective surface are more convincing if they actually appear to reflect the other geometry in the scene. The best-known method for this is the (generally) non-real-time method of recursive ray tracing, which is not currently suitable for general interactive systems. However, we can once again use a mixture of lightinglike computations and textures to create very convincing reflections.

Environment mapping is a technique that uses a texture or set of textures that represents an "inside looking out" representation of the entire scene in all directions. It can be thought of as a spherical or cube-shaped set of images that represent a panorama of the scene. These images can be statically drawn offline, or on modern systems can even be rendered every frame to better represent the current scene. The environment map can be thought of as infinitely

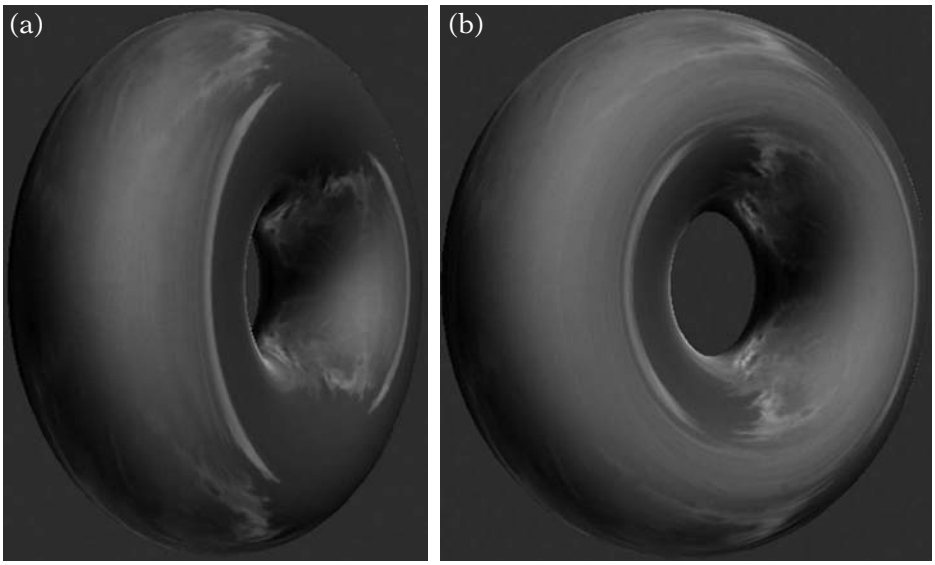


Figure 8.23 Environment mapping applied to simple objects.

large or infinitely distant. Thus, any normalized direction vector maps to a single, fixed location in the environment map.

Environment maps are applied to a surface dynamically — they are not sampled via a priori texture coordinates; the mapping of the environment map will change as the scene and the view changes. The most common method used is to compute the reflection of the view vector in a manner similar to that used for specular lighting earlier in this chapter. The reflected view vector represents the direction that the viewer sees in the reflective surface. By sampling the environment map in this view direction, we can apply what appears to be a convincing reflection to a surface with little more than a vector computation (per vertex or per fragment) and a texture lookup (per fragment). Figure 8.23 shows how this effect can be applied to a simple object. Note that in the two views of the object, the environment map moves like a reflection as the object rotates with respect to the viewer.

8.12 Shadows

Shadows are an extremely important component of real-world lighting. However, while we think of them as a form of lighting, the challenge in rendering accurate shadows has little in common with the per-vertex and per-fragment direct lighting formulae we discussed earlier in this chapter. In a sense, shadowing is much less about lighting than it is about occlusion or intersection. Diffuse and specular lighting formulae are concerned mainly with determining how the light incident upon a surface is reflected toward the viewer. Shadowing, on the other hand, is far more concerned with determining whether light from a given light source reaches the surface in the first place. Unlike surface lighting calculations, shadowing is dependent upon all of the objects that *might* interpose between the light source and the surface being lit.

A single surface shader is rarely if ever enough to implement shadowing. Shadowing is generally a multipass technique over the entire scene. The first pass involves determining which surfaces receive light from a given source, and the second pass involves applying this shadowing information to the shading of the objects in the scene. The many algorithms used to approximate real-time shadows differ in their approaches to both passes. With the advent of high-powered, programmable shading hardware, the push in shadowing methods over the past decade has focused on leveraging the rendering hardware as much as possible for both passes, avoiding expensive CPU-based computation. These algorithms have centered on the concept of using a first pass that involves rendering the scene from the point of view of the *light source*, as geometry visible from the point of view of the light is exactly the geometry that will be lit by that light source. Geometry that cannot be "seen" from a

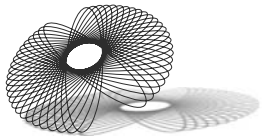
light's location is exactly the geometry that will be in shadow. Geometry that falls on the boundaries of these two cases is likely to be in the *penumbra*, or partially shadowed region, when rendering soft shadows.

Since the real core of shadowing methods lie in the structure of the two-pass algorithms rather than in the mathematics of lighting, the details of shadowing algorithms are beyond the scope of this book. A technique known as *ray tracing* (see Glassner [40]) uses ray–object intersection to track the way light bounces around a scene. Very convincing shadows (and reflections) could be computed using ray tracing, and the technique is very popular for non-real-time rendering. Owing to its computational complexity, this method is not generally used in real-time lighting (although modern shading languages and shading hardware are now capable of doing real-time ray tracing in some limited cases). Shadows are sometimes approximated using other tricks (see [1, 83, 84], or Chapter 13 of Eberly [25]). Excellent references for real-time shadows, both sharp and soft-edged, can be found in Fernando and Pharr [36, 92].

8.13 Chapter Summary

In this chapter we have discussed the basics of dynamic lighting, both in terms of geometric concepts and implementation using programmable shaders. Per-vertex and per-pixel lighting are very powerful additions to any 3D application, especially when mated with the use of multiple textures. Correct use of lighting can create compelling 3D environments at limited computational expense. As we have discussed, judicious use of lighting is important in order to maximize visual impact while minimizing additional computation.

For further information, there are numerous paths available to the interested reader. The growing wealth of shader resources includes websites [1, 84] and even book series [31]. Many of these new shaders are based on far more detailed and complex lighting models, such as those presented in computer graphics conference papers and journal articles like those of ACM SIGGRAPH or in books such as Watt and Policarpo [117].

Chapter 9

Rasterization

9.1 Introduction

The final major stage in the rendering pipeline is called *rasterization*. Rasterization is the operation that takes screen-space geometry, a fragment shader, and the inputs to that shader and actually draws the geometry to the low-level two-dimensional (2D) display device. Once again, we will focus on drawing sets of triangles, as these are the most common primitive in three-dimensional (3D) graphics systems. In fact, for much of this chapter, we will focus on drawing an individual triangle. For almost all modern display devices, this low-level "drawing" operation involves assigning color values to each and every dot, or *pixel*, on the display device.

At the conceptual level, the entire topic of rasterization is simply an implementation detail. Rasterization is required because the display devices we use today are based on a dense rectangular grid of light-emitting elements, or pixels (a short version of the term "picture elements"), each of whose colors and intensities are individually adjustable in every frame. For historical reasons relating to the way that picture tube–based televisions work, these displays are called *raster displays*.

Raster displays require that the images displayed on them be discretized into a rectangular grid of color samples for each image. In order to achieve this, a computer graphics system must convert the projected, colored geometry representations into the required grid of colors. Moreover, in order to render real-time animation, the computer graphics system must do so many times per second. This process of generating a grid of color samples from a projected scene is called *rasterization*.

By its very nature, rasterization is time consuming when compared to the other stages in the rendering pipeline. Whereas the other stages of the pipeline generally require per-object, per-triangle, or per-vertex computation, rasterization inherently requires computation of some sort for every pixel. At the time of this book's publication, displays 1,600 pixels wide by 1,200 pixels high — resulting in approximately *two million* pixels on the screen — are quite popular. Add to this the fact that rasterization will in practice often require each pixel to be computed several times, and we come to the realization that the number of pixels that must be computed generally outpaces the number of triangles in a given frame by a factor of 10, 20, or more.

Historically, in purely software 3D pipelines, it is not uncommon to see as much as 80 to 90 percent of rendering time spent in rasterization. This level of computational demand has led to the fact that rasterization was the first stage of the graphics pipeline to be accelerated via purpose-built consumer hardware. In fact, most 3D computer games began to *require* some form of 3D hardware by the early 2000s. This chapter will not detail the methods and code required to write a software 3D rasterizer, since most game developers no longer have a need to write them. For the details on how to write a set of rasterizers, see Hecker's excellent series of articles on perspective texture mapping in *Game Developer Magazine* [57].

Despite the fact that few if any game developers will need to implement even a subset of the rasterization pipeline themselves in a modern game, the topic of rasterization is still extremely relevant, even today. The basic concepts of rasterization lead to discussions of some of the most interesting and subtle mathematical and geometric issues in the entire rendering pipeline. Furthermore, an understanding of these fundamental concepts can allow a game developer to better understand why and how rendering artifacts and performance bottlenecks occur, even when the rasterization implementation is in dedicated hardware. Many of these basic concepts and low-level details can have visually relevant results in almost any 3D game. This chapter will highlight some of the fundamental concepts of rasterization that are most pivotal to a deeper understanding of the process of using a rendering system, either graphics processing unit (GPU) or computer processing unit (CPU) based.

9.2 Displays and Framebuffers

Every piece of display device hardware, whether it be a computer monitor, television, or some other such device, requires a source of image data. For computer graphics systems, this source of image data is called a *framebuffer* (so called because it is a buffer of data that holds the image information

for a "frame," or a screen's worth of image). In basic terms, a framebuffer is a 2D digital image: a block of memory that contains numerical values that represent colors at each point on the screen. Each color value represents the color of the screen at a given point—a *pixel*. Each pixel has red, green, and blue components. Put together, this framebuffer represents the image that is to be drawn on the screen. The display hardware reads these colors from memory every time it needs to update the image on the screen, generally at least 30 times per second and often 60 or more times per second.

As we shall see, framebuffers often include more than just a single color per pixel. While it is the final per-pixel color that is actually used to set the color and intensity of light emitted by each point on the display, the other per-pixel values are used internally during the rasterization process. In a sense, these other values are analogous to per-vertex normals and per-triangle material colors; while they are never displayed directly, they have a significant effect on how the final color is computed.

9.3 Conceptual Rasterization Pipeline

The steps required to rasterize an entire frame are shown in Figure 9.1. The first step is to clear out any previous image from the framebuffer. This can in

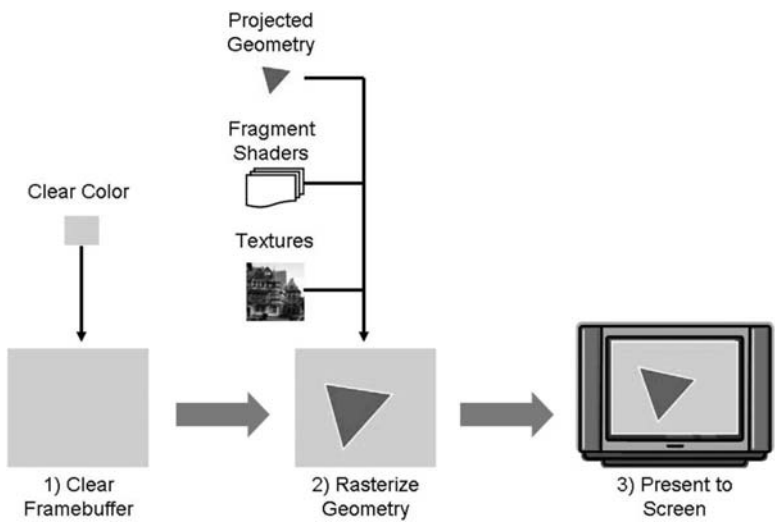


Figure 9.1 The steps to rasterizing a complete frame.

some cases be skipped; for example, if the scene geometry is known to cover the entire screen, then there is no need to clear the screen. The old image will be entirely overwritten by the new image in such a case. But for most applications, this step involves using the rendering application programming interface (API) to set all of the pixels in the framebuffer (in a single function call) to a fixed color.

The second step is to rasterize the geometry to the framebuffer. We will detail this stage in the rest of the chapter, as it is the most involved step of the three (by far).

The third step is to present the framebuffer image to the physical display. Most rendering is done to offscreen surfaces, to avoid flickering or other artifacts during rendering (specifically, to avoid having the user see a partially rendered frame). The "present" step makes the most recently rendered framebuffer visible to the screen. This stage is commonly known as *swapping* or *buffer swapping*, because historically it frequently involved (and in many cases still involves) "ping-ponging" between two buffers—drawing to one while the other is displayed, and then swapping the two buffers after each frame. However, other techniques described later in the chapter will require additional work to be done during the presentation step. Therefore, we will refer to this step by the more general term "present."

9.3.1 Rasterization Stages

There are several stages to even a simple rasterization pipeline. It should be noted that while these stages tend to exist in rasterization hardware implementations, hardware almost never follows the order (or even the structure) of the conceptual stages in the list that follows. This simple pipeline rasterizes a single triangle as follows:

1. Determine the visible pixels covered by the triangle.
2. Compute a color for the visible triangle at each such pixel.
3. Determine a final color for each pixel and write to the framebuffer.

The first stage further decomposes into two separate steps:

1. Determining the pixels covered by a triangle.
2. Determining which triangles are the ones visible at each pixel.

The rest of this chapter will discuss each of these pipeline stages in detail.

9.4 Determining the Fragments: Pixels Covered by a Triangle

9.4.1 Fragments

In order to progress any further in the rasterization phase of rendering, we must break triangles (or more generally, geometry) in screen space into pieces that more directly match the pixels in the framebuffer. This involves determining the intersection of pixel rectangles or pixel center points with a triangle. In the color and lighting chapters, we used the term *fragment* to represent an infinitesimal piece of surface area around a given point on a polygonal surface. Fragment shaders were described as being evaluated on these tiny pieces of surface.

At the rasterization level, fragments have a much more explicit but related definition. The result of the aforementioned process of breaking down screen-space triangles to match pixels is a set of what OpenGL calls "fragments" (the Direct3D documentation does not use this term, but the pipeline is analogous). These fragments can be thought of as pixel-sized pieces of a triangle in screen space. These can be visualized as a triangle "diced" into pieces by "cutting" along pixel boundaries. Many of these fragments (the interior of a triangle) will be square, the full size of the pixel square. We call these pixel-sized fragments *complete fragments*. However, along the edges of a triangle, these may be multisided polygons that fit inside of the pixel square and are thus smaller than a pixel. We call these smaller fragments *partial fragments*. In practice, these fragments may really be point samples of a triangle taken at the pixel center (similar to the concept we had of fragments in the lighting and shading chapters), but the basic idea is that fragments represent the pieces of a triangle that impinge upon a given pixel. We will think of pixels as being destinations or "bins" into which we place all of the fragments that cover the area of the pixel. As such, it is not a one-to-one mapping. A pixel may contain multiple fragments from different (or even the same) object, or a pixel may not contain any fragments in the current view of the scene.

The remainder of this chapter will use this more specific definition of fragments. Figure 9.2 shows a triangle overlaid with pixel rectangle boundaries. Figure 9.3 shows the same configuration broken into fragments, both complete and partial. The fragments are "exploded" slightly in the figure to better demonstrate the shapes of the partial fragments.

9.4.2 Depth Complexity

The number of fragments in an entire scene can be much smaller or much greater than the number of pixels on the screen. If only a subset of the

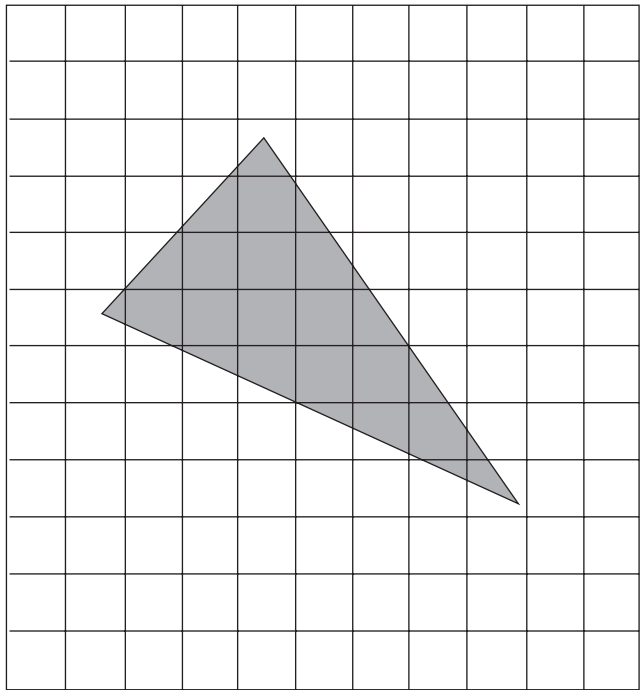


FIGURE 9.2 A screen-space triangle to be rasterized.

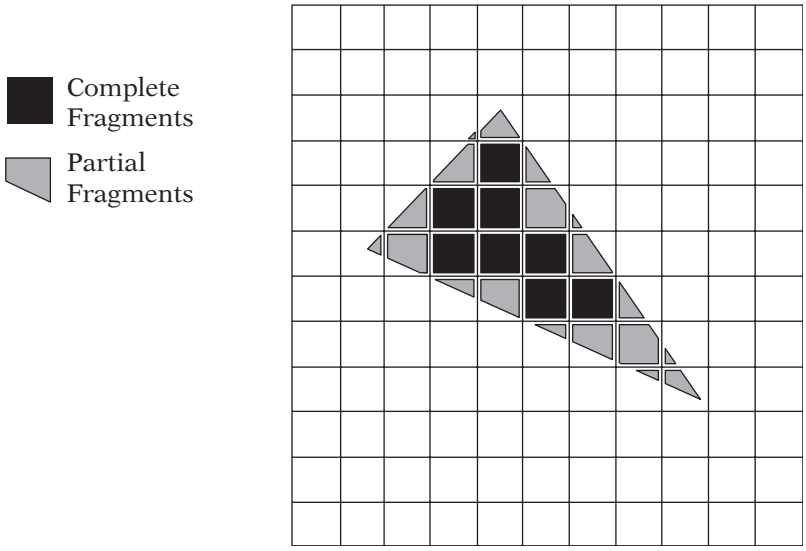


FIGURE 9.3 Fragments generated by the triangle. Complete fragments are dark gray; partial fragments are light gray.

screen is covered by geometry, then there may be many pixels that contain no fragments from the scene. On the other hand, if a lot of triangles overlap one another in screen space, then many pixels on the screen may contain more than one fragment. The ratio of the number of fragments in the scene in a given frame to the number of pixels on the screen is called the *depth complexity* or *overdraw*, because this ratio represents how many full screens worth of geometry comprises the scene. In general, scenes with a higher depth complexity are more expensive to rasterize. Note that this is an overall ratio for the whole view; a scene could have a depth complexity of two even if geometry only covers half of the screen. If, on average, the geometry on the half of the screen that is covered is four triangles deep, then the depth complexity would be two fragments per pixel amortized over the entire screen.

9.4.3 Converting Triangles to Fragments

Triangles are convex, no matter how they are projected by a projective transformation (in some cases, triangles may appear as a line or a point, but these are still convex objects). This is a very useful property, because it means that any triangle intersects a horizontal row of pixels (also called a *scan line*, for historical reasons having to do with CRT-based television displays) in at most one contiguous segment. Thus, for any scan line that intersects a triangle, we can represent the intersection with a single "span," a minimum x value and a maximum x value. Thus, the representation of a triangle during rasterization consists of a set of spans, one per scan line, that the triangle intersects. Furthermore, the convexity of triangles also implies that the set of scan lines intersected by a triangle is contiguous in y; there is a minimum and maximum y for a given triangle, which contains all of the nonempty spans. An example of the set of spans for a triangle is shown in Figure 9.4. The dark bands overlaid on the triangle represent the spans of adjacent fragments that will be used to draw the triangle.

The minimum y pixel coordinate for a triangle y_{min} is simply the minimum y value of the three triangle vertices. Similarly, the maximum y pixel coordinate y_{max} of the triangle is simply the maximum y value of the three vertices. Thus, a simple min/max computation among the three vertices defines the entire range of $(y_{max} - y_{min} + 1)$ spans that must be generated for a triangle.

The left-most and right-most fragments of each span may be partial fragments, since the edge of the triangle may not fall exactly on a pixel boundary. Also, the topmost and bottommost spans may contain partial fragments for the same reason. The remaining fragments for a triangle will be complete fragments.

Generating the spans themselves simply involves intersecting the horizontal scan line with the edges of the triangle. Owing to the convexity of the

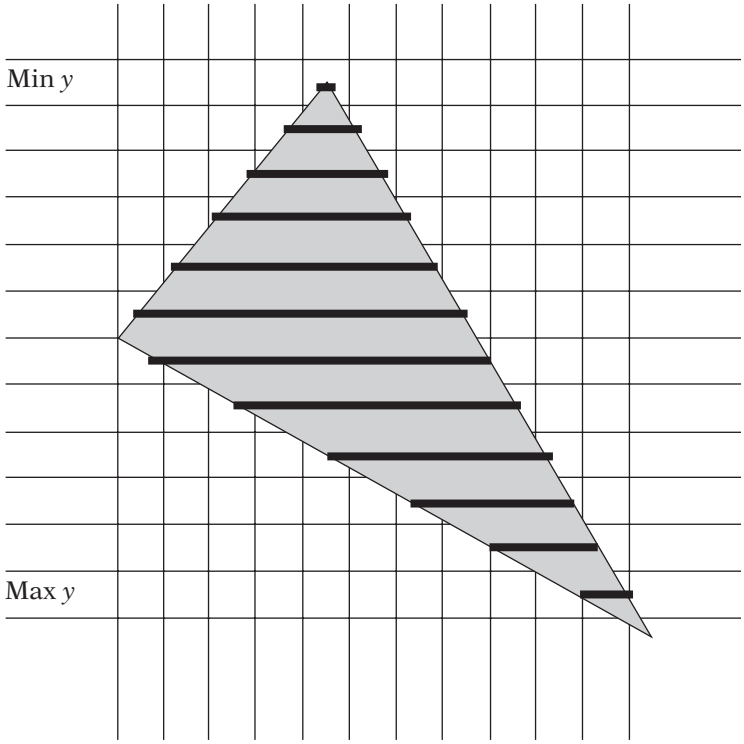


Figure 9.4 A triangle and its raster spans.

triangle, unless the scan line intersects a vertex, that scan line will intersect exactly two of the edges of the triangle: one to cross from outside the triangle into it, and one to leave again. These two intersection points will define the minimum and maximum x values of the span.

9.4.4 Handling Partial Fragments

Complete fragments always continue on to the next stage of the rasterization process. The fate of partial fragments, however, depends upon the particular rendering system. In more advanced systems, all partial fragments at a pixel are passed on as partial fragments, and the visibility and color of the final pixel may be influenced by all of them. However, simpler rasterization systems do not handle partial fragments, and must decide whenever a partial fragment is generated whether to drop the fragment or else promote it to a complete fragment. A common method for solving this is to keep partial fragments if and only if they contain the pixel's center point. This is sometimes called *point*

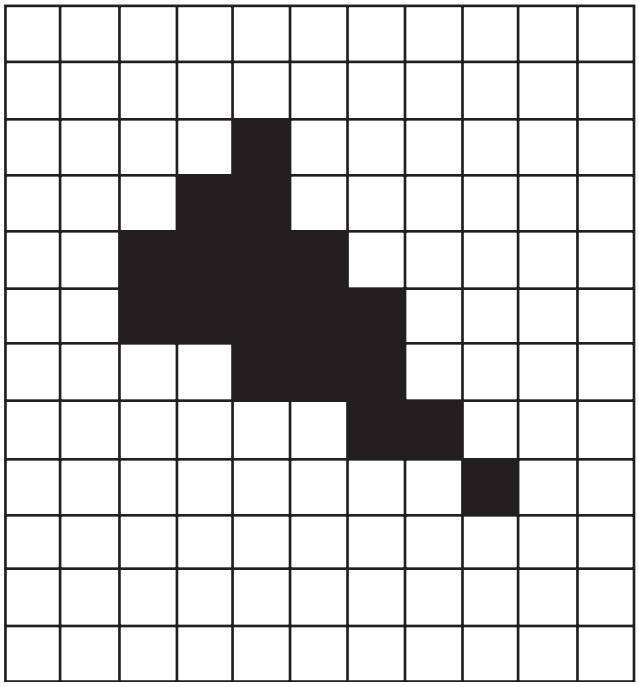


FIGURE 9.5 Fragments from Figure 9.3 rasterized using point sampling.

sampling of geometry, as an entire fragment is generated or not generated based on a single point sample within each pixel. Figure 9.5 shows the same triangle as in Figure 9.3, but with the partial fragments either dropped or promoted to complete fragments, based on whether the fragment contains the pixel's center point.

The behavior of such a graphics system when a triangle vertex or edge falls *exactly* on a pixel center is determined by a system-dependent *fill convention*, which ensures that if two triangles share a vertex or an edge, only one triangle will contribute a fragment to the pixel. This is very important, as without a well-defined fill convention, there may be "holes" (pixels where both triangles' partial fragments are dropped), or double-drawn pixels (where the partial fragments of both triangles are promoted to complete fragments) on the shared edges between triangles. Holes along a shared triangle edge allow the background color to show through what would otherwise be a continuous, opaque surface, making the surface appear to be "cracked." Double-drawn pixels along a shared edge result in more subtle artifacts, normally seen only when transparency or other forms of blending are used (see Section 9.9.1, Pixel Blending, later in this chapter). For details on implementing

point-sampled fill conventions, see Hecker's *Game Developer Magazine* article series [57].

9.5 Determining Visible Geometry

The overall goal in rendering geometry is to ensure that the final rendered images convincingly represent the given scene. At the highest level, this means that objects must appear to be correctly obscured by closer objects and must not be obscured by more distant objects. This process is known as *visible surface determination* (VSD), and there are numerous, very different ways of accomplishing it. The methods all involve comparing the depth of surfaces at one level of granularity or another and rendering them in such a way that the object of minimum depth (i.e., the closest object) at a given pixel is the one rendered to the screen.

Historically, numerous different methods have been used for VSD. Many of the early algorithms were based on clever sorting tricks, involving ordering the geometry back to front prior to rasterization. This was an expensive proposition normally computed per frame on the CPU. By far, the most common method in use today is a rasterization-based method: the depth buffer. Rasterizers were the first parts of the graphics pipeline to be accelerated with purpose-built hardware, meaning that a rasterizer-based visible surface determination system could achieve high performance. The depth buffer is also known as a *z-buffer*, which is actually a specific, special case of the more general depth buffering.

9.5.1 Depth Buffering

Depth buffering is based on the concept that visibility should be output-focused. In other words, since pixels are the final destination of our rendering pipeline, visibility should be computed on a per-pixel (or rather, per-fragment) basis. If the final color seen at each pixel is the color of the fragment with the minimum depth (of all fragments drawn to that pixel), the scene will appear to be drawn correctly. In other words, of all the fragments drawn to a pixel, the fragment with minimum depth should "win" the pixel and select that pixel's color. For the purposes of this discussion, we assume point-sampled geometry (i.e., there are no partial fragments).

Since common rasterization methods tend to render a triangle at a time, a given pixel may be redrawn several times over the course of a frame by fragments from different triangles. If we wish to avoid sorting the triangles by depth (and we do), then the fragment that should win a given pixel may not be the last one drawn to that pixel. We must have some method of storing

the depth of the current nearest fragment at each pixel, along with the color of that fragment.

Having stored this information, we can compute a simple test each time a fragment is drawn to a pixel. If the new fragment's depth is closer than the currently stored depth value at that pixel, then the new fragment wins the pixel. The color of the new fragment is computed and this new fragment color is written to the pixel. The fragment's depth value replaces the existing depth value for that pixel. If the new fragment has greater depth than that of the current fragment coloring the pixel, then the new fragment's color and depth are ignored, as the fragment represents a surface that is behind the closest known surface at the current pixel. In this case, we know that the new fragment will be obscured at that pixel, because we have already seen a fragment at that pixel that is closer than the newest fragment. Figure 9.6

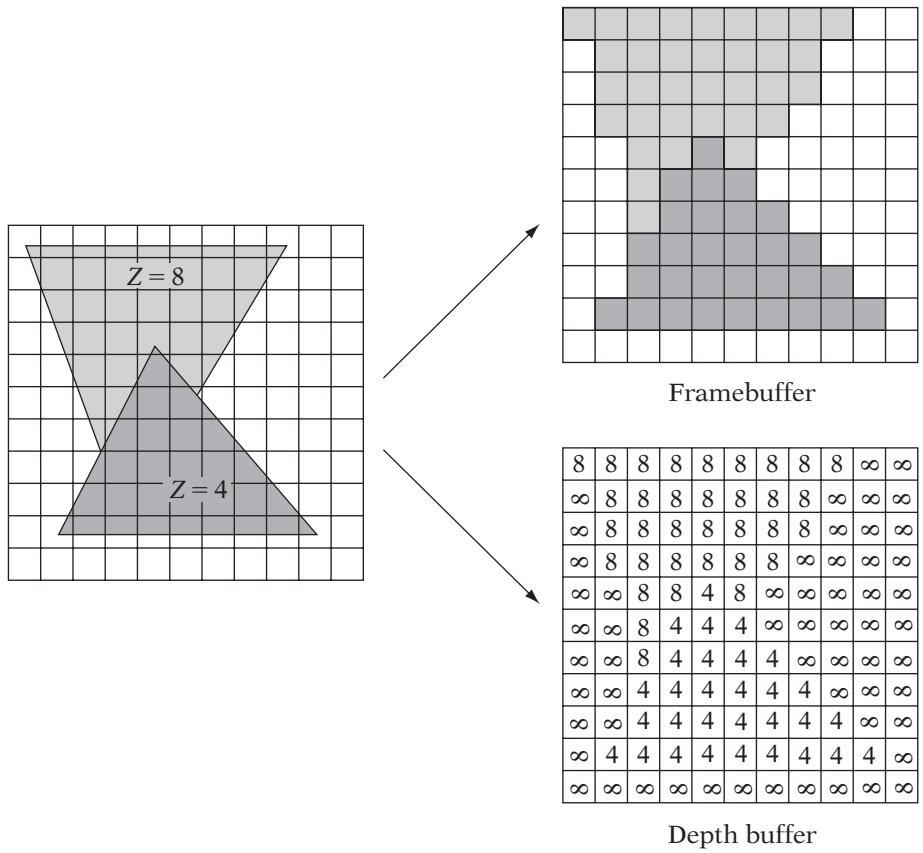


FIGURE 9.6 Two triangles rendered to a depth buffer.

represents the rendering of the fragments from two triangles to a small depth buffer. Note how the closer triangle's fragment always wins the pixel (the correct result), even if it is drawn first.

Because the method is per-pixel and thus per fragment, the depth of each triangle is computed on a per-fragment granularity, and this value is used in the depth comparison. As a result of this finer subtriangle granularity, the depth buffer automatically handles triangle configurations that cannot be correctly displayed using per-triangle sorting. Geometry may be passed to the depth buffer in any order. The situation in which this random order can be problematic is when two fragments at a given pixel have equal depth. In this case, order will matter, depending on the exact comparison used to order depth (i.e., $<$ or $\leq$). However, such circumstances are problematic with almost any visible surface method.

There are several drawbacks to the depth buffer, although most of these are no longer significant on modern PCs or game consoles. One of the historical drawbacks of the depth buffering method is implied in the name of the method; it requires a *buffer* or array of depth values, one per pixel. This is a large block of memory, often requiring as much memory as the framebuffer itself. Also, just as the framebuffer must be cleared to the background color before each frame, the depth buffer must be cleared to the background depth, which is generally the maximum representable depth value. These issues can be significant on handheld and embedded 3D systems, where GPU memory is limited. Finally (and still relevant on PCs and consoles), the depth buffer requires the following work for *each fragment*:

- Computation of a depth value for the fragment.
- Lookup of the existing pixel depth in the depth buffer.
- Comparison of these two values.
- (For new "winner" fragments only): Writing the new depth to the depth buffer.

This additional work per fragment makes depth buffering unsuitable for constant use in most software rasterizers. Fully software 3D systems tend to use optimized geometry sorting wherever possible, reserving depth buffering for the few objects that truly require it. One example was early third-person-shooter game rendering engines, which put enormous work into specialized sorting of the environments or "levels," avoiding any depth buffer testing for them. This left enough CPU cycles to render the animated characters, monsters, and small objects (which covered far fewer pixels than the scenery) using software depth buffering.

In addition, the depth buffer does not fix the potential performance problems of high-depth complexity scenes. We must still compute the depth of

every fragment and compare it to the buffer. However, it can make overdraw less of an issue in some cases, since it is not necessary to compute or write the color of any fragment that fails the depth test. In fact, some applications will try to render their depth-buffered scenes in roughly near-to-far ordering (while still avoiding per-triangle, per-frame sorting on the CPU) so that the later geometry is likely to fail the depth buffer test and not require color computations.

Depth buffering is extremely popular in 3D applications that run on hardware-accelerated platforms, as it is easy to use and requires little application code or host CPU computation and produces quality images at high performance.

Computing Per-Fragment Depth Values

The first step in computing the visibility of a fragment using a depth buffer is to compute the depth value of the current fragment. As we shall see, z_{ndc} (which appeared to be a rather strange choice for z back in Chapter 6) will work quite well. However, the reason why z_{ndc} works well and z_{view} does not is rather interesting.

In order to better understand the nature of how depth values change across a triangle in screen space, we must be able to map a point on the screen to the point in the triangle that projected to it. This is very similar to picking, and we will use several of the concepts we first discussed in Chapter 6. Owing to the nonlinear nature of perspective projection, we will find that our mapping from screen-space pixels to view-space points on a given triangle is somewhat complicated. We will follow this mapping through several smaller stages.

A triangle in view space is simply a convex subset of a plane in view space. As a result, we can define the plane of a triangle in view space by a normal vector to the plane $\hat{\mathbf{n}} = (a, b, c)$ and a constant d, such that the points $P = (x_p, y_p, z_p)$ in the plane are those that satisfy

$$\begin{aligned} ax_p + by_p + cz_p + d &= 0 \\ (a, b, c) \cdot (x_p, y_p, z_p) + d &= 0 \\ \hat{\mathbf{n}} \cdot (x_p, y_p, z_p) + d &= 0 \end{aligned} \tag{9.1}$$

Looking back at picking, a point in 2D normalized device (ND) coordinates (x_{ndc}, y_{ndc}) maps to the view-space ray $t\,\mathbf{r}$ such that

$$t\,\mathbf{r} = (x_{ndc}, y_{ndc}, -dist)t, \quad t \geq 0$$

where *dist* is the projection distance (the distance from the view-space origin to the projection plane). Any point in view space that projects to the pixel at

(x_{ndc}, y_{ndc}) must intersect this ray. Normally, we cannot "invert" the projection matrix, since a point on the screen maps to a ray in view space. However, by knowing the plane of the triangle, we can intersect the triangle with the view ray as follows. All points P in view space that fall in the plane of the triangle are given by equation 9.1. In addition, we know that the point on the triangle that projects to (x_{ndc}, y_{ndc}) must be equal to $t\,\mathbf{r}$ for some t. Substituting the vector $t\,\mathbf{r}$ for the points (x_p, y_p, z_p) in equation 9.1 and solving for t,

$$\begin{aligned} \hat{\mathbf{n}} \cdot (t\,\mathbf{r}) + d &= 0 \\ t(\hat{\mathbf{n}} \cdot \mathbf{r}) &= -d \\ t &= \frac{-d}{\hat{\mathbf{n}} \cdot \mathbf{r}} \end{aligned}$$

From this value of t, we can compute the point along the projection ray $(x_{view}, y_{view}, z_{view}) = t\,\mathbf{r}$ that is the view-space point on the triangle that projects to (x_{ndc}, y_{ndc}). This amounts to finding

$$\begin{aligned} (x_{view}, y_{view}, z_{view}) &= t\,\mathbf{r} \\ &= t(x_{ndc}, y_{ndc}, -dist) \\ &= \frac{-d(x_{ndc}, y_{ndc}, -dist)}{\hat{\mathbf{n}} \cdot \mathbf{r}} \\ &= \frac{-d(x_{ndc}, y_{ndc}, -dist)}{\hat{\mathbf{n}} \cdot (x_{ndc}, y_{ndc}, -dist)} \\ &= \frac{-d(x_{ndc}, y_{ndc}, -dist)}{\hat{\mathbf{n}}_x x_{ndc} + \hat{\mathbf{n}}_y y_{ndc} - \hat{\mathbf{n}}_z dist} \end{aligned} \tag{9.2}$$

However, we are only interested in z_{view} right now, since we are trying to compute a per-fragment value for depth buffering. The z_{view} component of equation 9.2 is

$$z_{view} = \frac{dist\ d}{\hat{\mathbf{n}}_x x_{ndc} + \hat{\mathbf{n}}_y y_{ndc} - \hat{\mathbf{n}}_z dist} \tag{9.3}$$

As a quick check of a known result, note that in the special case of a triangle of constant depth $z_{view} = z_{const}$, we can substitute

$$\hat{\mathbf{n}} = (0, 0, 1)$$

and

$$d = -z_{const}$$

Substituted into equation 9.3 this evaluates to the expected constant $z_{view} = z_{const}$:

$$\begin{aligned} z_{view} &= \frac{dist(-z_{const})}{0x_{ndc} + 0y_{ndc} - 1dist} \\ &= \frac{-dist\ z_{const}}{-dist} \\ &= z_{const} \end{aligned}$$

As defined in equation 9.3, z_{view} is an expensive value to compute per fragment (in the general, nonconstant depth case), because it is a fraction with a nonconstant denominator. This would require a per-fragment division to compute z_{view}, which is more expensive than we would like. However, depth buffering requires only the ability to compare depth values against one another. If we are comparing z_{view} values, we know that they decrease with increasing depth (as the view direction is $-z$), giving a depth test of

$$\begin{aligned} z_{view} &\geq DepthBuffer \rightarrow \text{New fragment is visible} \\ z_{view} &< DepthBuffer \rightarrow \text{New fragment is not visible} \end{aligned}$$

However, if we compute and store the reciprocal (the multiplicative inverse) of z_{view}, then a similar comparison still works in the same manner. If we use the reciprocal of all of the z_{view} values, we get

$$\begin{aligned} \frac{1}{z_{view}} &\leq DepthBuffer \rightarrow \text{New fragment is visible} \\ \frac{1}{z_{view}} &> DepthBuffer \rightarrow \text{New fragment is not visible} \end{aligned}$$

If we reciprocate equation 9.3, we can see that the per-fragment computation becomes simpler:

$$\begin{aligned} \frac{1}{z_{view}} &= \frac{\hat{\mathbf{n}}_x x_{ndc} + \hat{\mathbf{n}}_y y_{ndc} - \hat{\mathbf{n}}_z dist}{dist\ d} \\ &= \left(\frac{\hat{\mathbf{n}}_x}{dist\ d}\right) x_{ndc} + \left(\frac{\hat{\mathbf{n}}_y}{dist\ d}\right) y_{ndc} - \left(\frac{\hat{\mathbf{n}}_z d}{dist\ d}\right) \end{aligned}$$

where all of the parenthesized terms are constant across a triangle. In fact, this forms an affine mapping of ND coordinates to $1/z_{view}$. Since we know that there is an affine mapping from pixel coordinates (x_s, y_s) to ND coordinates (x_{ndc}, y_{ndc}), we can compose these affine mappings into a single affine mapping

from screen-space pixel coordinates to $1/z_{view}$. As a result, for a given projected triangle,

$$\frac{1}{z_{view}} = fx_s + gy_s + h$$

where f, g, and h are real values and are constant per triangle. We define the preceding mapping for a given triangle as

$$InvZ(x_s, y_s) = fx_s + gy_s + h$$

An interesting property of $InvZ(x_s, y_s)$ (or of any affine mapping, for that matter) can be seen from the derivation

$$\begin{aligned} InvZ(x_s + 1, y_s) - InvZ(x_s, y_s) &= (f(x_s + 1) + gy_s + h) - (fx_s + gy_s + h) \\ &= f(x_s + 1) - (fx_s) \\ &= f \end{aligned}$$

meaning that

$$InvZ(x_s + 1, y_s) = InvZ(x_s, y_s) + f$$

and similarly

$$InvZ(x_s, y_s + 1) = InvZ(x_s, y_s) + g$$

In other words, once we compute our $InvZ$ depth buffer value for any "base" fragment, we can compute the depth buffer value of the next fragment in the span by simply adding f. Once we compute a base depth buffer value for a given span, as we step along the scan line, filling the span, all we need to do is add f to our current depth between each adjacent fragment (Figure 9.7). This makes the per-fragment computation of a depth value very fast indeed. In fact, once the base $InvZ$ of the first span is computed, we may add or subtract f and g to or from the previous span's base depth to compute the base depth of the next span. This technique is known as *forward differencing*, as we use the difference (or delta) between the value at a fragment and the value at the next fragment to step along, updating the current depth. This method will work for *any* value for which there is an affine mapping from screen space. We refer to such values as *affine in screen space*, or *screen affine*.

In fact, we can use the z_{ndc} value that we computed during projection as a replacement for $InvZ$. In Chapter 6, on viewing and projection, we computed

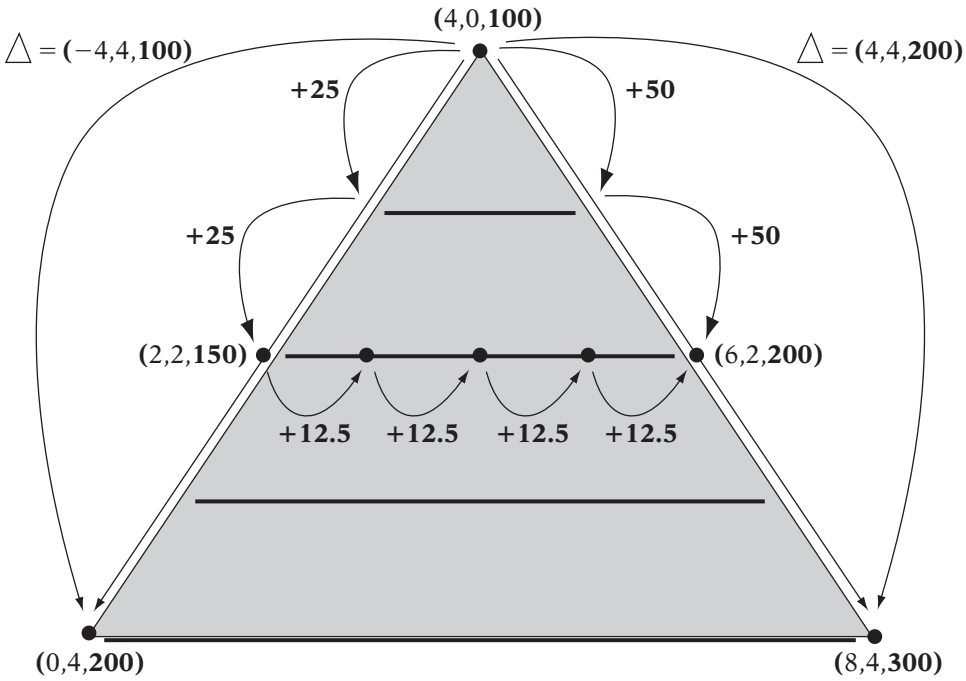


FIGURE 9.7 Forward differencing the depth value.

a z_{ndc} value that is equal to -1 at the near plane and 1 at the far plane and was of the form

$$z_{ndc} = \frac{a + bz_{view}}{z_{view}} = a\frac{1}{z_{view}} + b$$

which is an affine mapping of $InvZ$. As a result, we find that our existing value z_{ndc} is screen affine and is suitable for use as a depth buffer value. This is the special case of depth buffering we mentioned earlier, often called *z-buffering*, as it uses z_{ndc} directly.

Numerical Precision and *Z*-Buffering

In practice, depth buffering in screen space has some numerical precision limitations that can lead to visual artifacts. As was mentioned earlier in the discussion of depth buffers, the order in which objects are drawn to a depth buffering system (at least in the case of opaque objects) is only an issue if the depth values of the two surfaces (two fragments) are equal at a given pixel. In theory, this is unlikely to happen unless the geometric objects in question are truly coplanar. However, because computer number representations do not

have infinite precision (recall the discussion in Chapter 1), surfaces that are not coplanar can map to the same depth value. This can lead to objects being drawn in the wrong order.

If our depth values were mapped linearly into view space, then a 16-bit, fixed-point depth buffer would be able to correctly sort any objects whose surfaces differed in depth by about 1/60,000 of the difference between the near and far plane distances. This would seem to be more than enough for almost any application. For example, with a view distance of 1 km, this would be equal to about 1.5 cm of resolution. Moving to a higher-resolution depth buffer would make this value even smaller.

However, in the case of z-buffering, representable depth values are not evenly distributed in view space. In fact, the depth values stored to the buffer are basically $1/Z_{view}$, which is definitely not an even distribution of view space Z. A graph of the depth buffer value over view space Z is shown in Figure 9.8. This is a hyperbolic mapping of view space Z into depth buffer values — notice how little the depth value changes with a change in Z toward the far plane. Using a fixed-point value for this leads to very low precision in the distance, as large intervals of Z map to the same fixed-point value of inverse Z. In fact, a common estimate is that a z-buffer focuses 90 percent of its precision in the closest 10 percent of view space Z. This means that the fragments of distant objects are often sorted incorrectly with respect to one another.

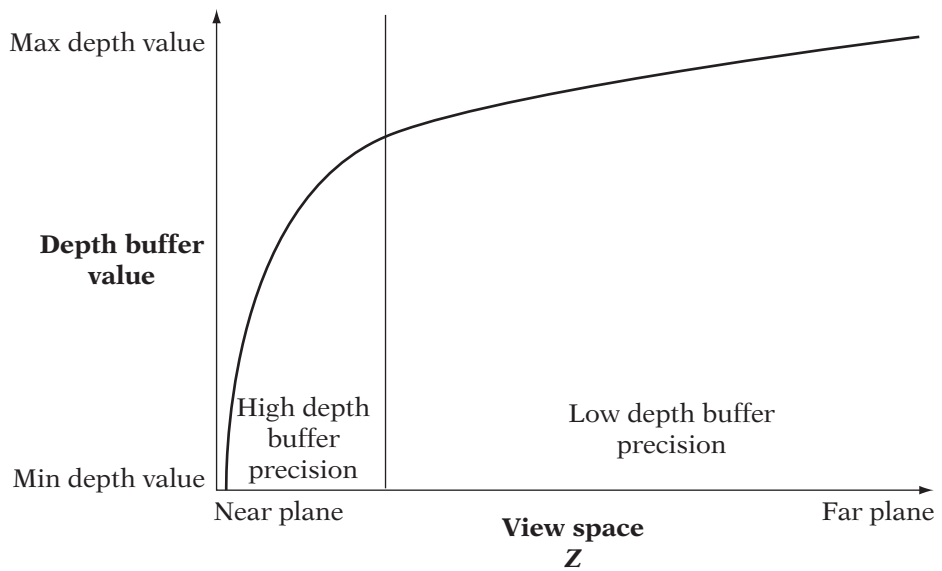


Figure 9.8 Depth buffer value as a function of view space Z.

The simplest way to avoid these issues is to maximize usage of the depth buffer by moving the near plane as far out as possible so that the accuracy close to the near plane is not wasted. Another method that is popular in 3D hardware is known as the w-buffer. The w-buffer interpolates a screen-affine value for depth (often $1/w$) at a high precision, then computes the inverse of the interpolation at each pixel to produce a value that is linear in view space (i.e., $1/\frac{1}{w}$). It is this inverted value that is then stored in the depth buffer. By quantizing (dropping the extra precision used during interpolation) and storing a value that is linear in view space, the hyperbolic nature of the z-buffer can be avoided to some degree.

Finally, floating-point depth buffers are available on some platforms. These can be particularly useful when the depth-buffered depth values are remapped such that the depth values map to 1.0 at the near plane and 0.0 at the far plane. In this case, the natural precision characteristics of floating-point numbers can be used to counteract *some* of the hyperbolic nature of z-buffer values. Actually, floating-point depth buffers can have other issues, overcorrecting and leaving the region of the scene closest to the camera with *too little* precision. This is particularly noticeable in rendered scenes because the geometry nearest the camera is the most obvious to the viewer. All of these methods have scene- and application-dependent trade-offs.

9.5.2 Depth Buffering in Practice

Using depth buffering in most graphics systems requires additions to several points in rendering code:

- Creation of the depth buffer when the framebuffer is created.
- Clearing the depth buffer each frame.
- Enabling depth buffer testing and writing.

The first step is to ensure that the rendering window or device is created with a depth buffer. This differs from API to API, with `Iv` automatically allocating a depth buffer in all cases. Having requested the creation of a depth buffer (and in most cases, it is just that — a *request* for a depth buffer, dependent upon hardware support), the buffer must be cleared at the start of each frame. The depth buffer is generally cleared using the same function as the framebuffer clear. `Iv` uses the `IvRenderer` function, `ClearBuffers`, but with a new argument, `kDepthClear`. While the depth buffer can be cleared independently of the framebuffer using

```
renderer->ClearBuffers(kDepthClear);
```

if you are clearing both buffers at the start of a frame, it can be faster on some systems to clear them both with a single call, which is done as follows in `Iv`:

```
renderer->ClearBuffers(kColorDepthClear);
```

To enable or disable depth testing we simply set the desired test mode using the `IvRenderer` function `SetDepthTest`. To disable testing, pass `kDisableDepthTest`. To enable testing, pass one of the other test modes (e.g., `kLessDepthTest`). By default, depth testing is disabled, so the application should enable it explicitly prior to rendering. The most common depth testing modes are `kLessDepthTest` and `kLessEqualDepthTest`. The latter mode causes a new fragment to be used if its depth value is less than or equal to the current pixel depth.

The writing of depth values also can be enabled or disabled, independent of depth testing. As we shall see later in this chapter, it can be useful to enable depth testing while disabling depth buffer writing. A call to the `IvRenderer` function `SetDepthWrite` can enable or disable writing the z-buffer.

9.6 Computing Fragment Shader Inputs

The next stage in the rasterization pipeline is to compute the overall color (and possibly other shader output values) of a fragment by evaluating the currently active fragment shader for the current fragment. This in turn requires that the input values used by the shader be evaluated at the current fragment location. These inputs come in numerous forms, as discussed in the previous two chapters. Common sources include:

- Per-object uniform values set by the application.
- Per-vertex attributes generated or passed through from the source vertices by the vertex shader.
- Indirect per-fragment values, generally from textures.

Note that as we saw in the lighting chapter (Chapter 8), numerous sources may exist for a given fragment. Each of them must be independently evaluated per-fragment as a part of shader input source generation. Having computed the per-fragment source values, a final fragment color must be generated by running the fragment shader. Chapter 8 discussed various ways that per-fragment (also referred to as "per-sample" in Chapter 8) vertex color values, per-vertex lighting values, and texture colors can be combined in the fragment shader. The shader generates a final fragment color that is passed to the last

stage of the rasterization pipeline, blending (which will be discussed later in this chapter).

The next few sections will discuss how shader source values are computed per fragment from the sources we have listed. While there are many possible methods that may be used, we will focus on methods that are fast to compute in screen space and are well suited to the scan line–centric nature of most rasterizer software and even some rasterizer hardware.

9.6.1 Uniform Values

As with all other stages in the pipeline, per-object values or colors are the easiest to rasterize. For each fragment, the constant uniform value may be passed down to the shader directly. No per-fragment evaluation or computation is required. As a result, uniform values can have minimal performance impact to the fragment shading process.

9.6.2 Per-Vertex Attributes

Per-vertex values that are either passed through the vertex shader into the fragment shader or computed in the vertex shader and then passed to the fragment shader are referred to in OpenGL as *varying values*. These values are defined only at the three vertices of each triangle, and thus must be interpolated to determine a value at each fragment center in the triangle. As discussed in the shading and lighting chapters, this is generally done by linearly interpolating between the three vertex values in object space. As we shall see, in the general case this can be an expensive operation to compute correctly for each of a triangle's fragments. However, we will first look at the special case of triangles of constant depth. The mapping in this case is not at all computationally expensive, making it a tempting approximation to use even when rendering triangles of nonconstant depth (especially in a software renderer).

To analyze the constant-depth case, we will determine the nature of the mapping of our constant-depth triangle from pixel space, through NDC space, into view space, through barycentric coordinates, and finally to the per-vertex source attributes. We start first with a special case of the mapping from pixel space to view space.

The overall projection equations derived in Chapter 6 (mapping from view space through NDC space to pixel coordinates) were all of the form

$$x_s = \frac{ax_{view}}{z_{view}} + b$$
$$y_s = \frac{cy_{view}}{z_{view}} + d$$

where both $a, c \neq 0$. If we assume that a triangle's vertices are all at the same depth (i.e., view space Z is equal to a constant z_{const} for all points in the triangle), then the projection of a point in the triangle is

$$x_s = \frac{ax_{view}}{z_{const}} + b = \left(\frac{a}{z_{const}}\right) x_{view} + b = a' x_{view} + b$$
$$y_s = \frac{cy_{view}}{z_{const}} + d = \left(\frac{c}{z_{const}}\right) y_{view} + d = c' y_{view} + d$$

Note that $a, c \neq 0$ implies that $a', c' \neq 0$, so we can rewrite these such that

$$x_{view} = \frac{x_s - b}{a'}$$
$$y_{view} = \frac{y_s - d}{c'}$$

Thus, for triangles of constant depth z_{const}:

- Projection forms an affine mapping from screen vertices to view-space vertices on the $z_{view} = z_{const}$ plane.
- Barycentric coordinates are an affine mapping of view-space vertices (as we saw in Chapter 2).
- Vertex colors define an affine mapping from a barycentric coordinate to a color (Gouraud shading, as seen in Chapter 7).

If we compose these affine mappings, we end up with an affine mapping from screen-space pixel coordinates to color. We can write this affine mapping from pixel coordinates to color as

$$Color(x_s, y_s) = C_x x_s + C_y y_s + C_0$$

where C_x, C_y, and C_0 are all colors (each of which are possibly negative or greater than 1.0). For a derivation of the formula that maps the three screen-space pixel positions and corresponding trio of vertex colors to the three colors C_x, C_y, and C_0, see page 126 of Eberly [25]. From our earlier derivation of the properties of inverse Z in screen space, we note that $Color(x_s, y_s)$ is screen affine *for triangles of constant z*:

$$\begin{aligned} Color(x_s + 1, y_s) - Color(x_s, y_s) &= (C_x(x_s + 1) + C_y y_s + C_0) - (C_x x_s + C_y y_s + C_0) \\ &= C_x(x_s + 1) - (C_x x_s) \\ &= C_x \end{aligned}$$

meaning that

$$Color(x_s + 1, y_s) = Color(x_s, y_s) + C_x$$

and similarly

$$Color(x_s, y_s + 1) = Color(x_s, y_s) + C_y$$

As with inverse Z, we can compute per-fragment values for per-vertex attributes for a constant-z triangle simply by computing forward differences of the color of a "base fragment" in the triangle.

When a triangle that does not have constant depth in camera space is projected using a perspective projection, the resulting mapping is not screen affine. From our discussion of depth buffer values, we can see that given a general (not necessarily constant-depth) triangle in view space, the mapping from NDC space to the view-space point on the triangle is of the form

$$x_{view} = \frac{dx_{ndc}}{ax_{ndc} + by_{ndc} + c}$$

$$y_{view} = \frac{d'y_{ndc}}{ax_{ndc} + by_{ndc} + c}$$

$$z_{view} = \frac{d''}{ax_{ndc} + by_{ndc} + c}$$

These are projective mappings, not affine mappings as we had in the constant-depth case. This means that the overall mapping from screen space to linearly interpolated per-vertex attributes is also projective. Such a projective mapping requires two forward differences (one for the numerator and one for the denominator) and a division per-attribute component (i.e., 3 for an RGB color) *per fragment*. In order to correctly interpolate vertex attributes of a triangle in perspective, we must use this more complex projective mapping.

Most hardware rendering systems now interpolate all per-vertex attributes in a perspective-correct manner. However, this has not always been universal, and in the case of older software rendering systems running on lower-powered platforms, it was too expensive. If the per-vertex attributes being interpolated are colors from per-vertex lighting, such as in the case of Gouraud shading, it is possible to make an accuracy-speed trade-off. Keeping in mind that Gouraud shading is an approximation method in the first place, there is somewhat decreased justification for using the projective mapping on the basis of "correctness." Furthermore, Gouraud-shaded colors tend to interpolate so smoothly that it can be difficult to tell whether the interpolation is perspective correct or not. In fact, Heckbert and Moreton [56] mention that the New York

Institute of Technology's offline renderer interpolated colors incorrectly in perspective for several years before anyone noticed! As a result, software graphics systems have often avoided the expensive, perspective-correct projective interpolation of Gouraud colors and have simply used the affine mapping and forward differencing. However, our next interpolant, texture coordinates, will not be so forgiving of issues in perspective-correct interpolation.

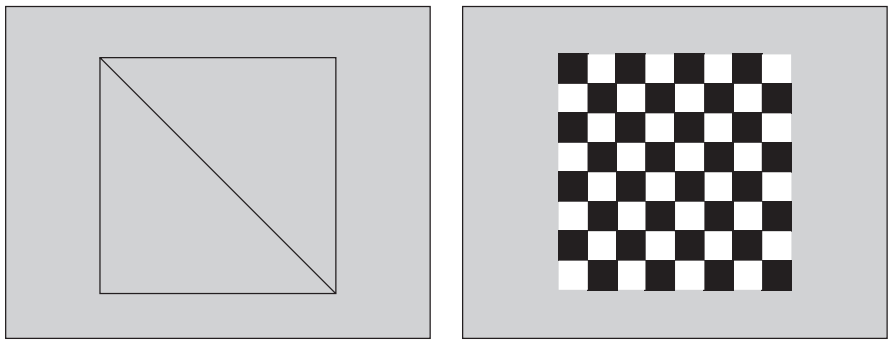
9.6.3 Interpolating Texture Coordinates

The process of rasterizing a texture starts by interpolating the per-vertex texture coordinates to determine the correct value at each fragment. Actually, it is generally the texel coordinates (the texture coordinates multiplied by the texture image dimensions) that are interpolated in a rasterizer. This process is analogous to interpolating other per-vertex attributes. However, because texture coordinates are actually used somewhat differently than vertex colors in the fragment shader, we are not able to use the screen-affine approximation described previously. Texture coordinates require the correct perspective interpolation. The indirect nature of texture coordinates means that while the texture coordinates change smoothly and subtly over a triangle, the resulting texture color lookup does not.

The issue in the case of texture coordinates has to do with the properties of affine and projective transformations. Affine transformations map parallel lines to parallel lines, while projective transformations guarantee only to map straight lines to straight lines. Anyone who has ever looked down a long, straight road knows that the two lines that form the edges of the road appear to meet in the distance, even though they are parallel. Perspective, being a projective mapping, does not preserve parallel lines.

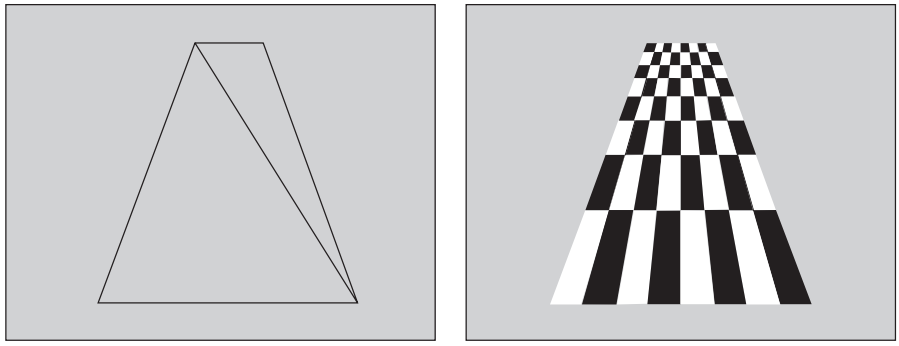
The classic example of the difference between affine and projective interpolations of texture coordinates is the checkerboard square, drawn in perspective. Figure 9.9 shows a checkered texture as an image, along with the image applied with wrapping to a square formed by two triangles (the two triangles are shown in outline, or *wire frame*). When the top is tilted away in perspective, note that if the texture is mapped using a projective mapping (Figure 9.10), the vertical lines converge into the distance as expected.

If the texture coordinates are interpolated using an affine mapping (Figure 9.11), we see two distinct visual artifacts. First, within each triangle, all of the parallel lines remain parallel, and the vertical lines do not converge the way we expect. Furthermore, note the obvious "kink" in the lines along the square's diagonal (the shared triangle edge). This might at first glance seem to be a bug in the interpolation code, but a little analysis shows that it is actually a basic property of an affine transformation. An affine transformation is defined by the three points of a triangle. As a result, having defined the three points of the triangle and their texture coordinates, there are no more



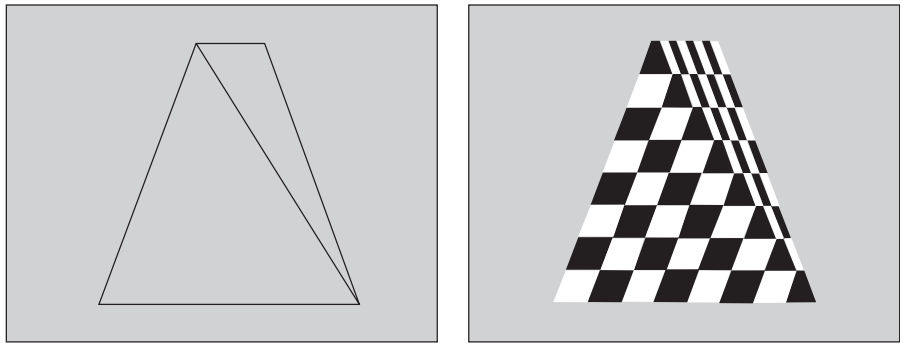
Wire-frame view Textured view

FIGURE 9.9 Two textured triangles parallel to the view plane.



Wire-frame view Textured view

FIGURE 9.10 Two textured triangles oblique to the view plane, drawn using a projective mapping.



Wire-frame view Textured view

FIGURE 9.11 Two textured triangles oblique to the view plane, drawn using an affine mapping.

degrees of freedom in the transformation. Each triangle defines its transform independent of the other triangles, and the result is a bend in what should be a set of lines across the square.

The projective transform, however, has additional degrees of freedom, represented by the depth values associated with each vertex. These depth values change the way the texture coordinate is interpolated across the triangle and allow straight lines in the mapped texture image to remain straight on-screen, even across the triangle boundaries. The downside of this projective mapping is that it requires the following operations per fragment for correct evaluation:

1. An affine forward difference operation to update the numerator for u_{texel}.
2. An affine forward difference operation to update the numerator for v_{texel}.
3. An affine forward difference operation to update the shared denominator (both u_{texel} and v_{texel} can use the same denominator, as it is based on inverse depth of the triangle at the pixel).
4. A division to recover the perspective-correct u_{texel}.
5. A division to recover the perspective-correct v_{texel}.

While many PC games and some video game consoles in the 1990s used less expensive (and less correct) approximations of true perspective texturing, on modern hardware rasterization systems, per-fragment perspective-correct texturing is simply assumed. Also, the fact that programmable fragment shaders can allow basically any per-vertex attribute to be used as a texture coordinate has further influenced hardware vendors in the move to interpolate all vertex attributes in correct perspective.

9.6.4 Other Sources of Texture Coordinates

Direct use of per-vertex texture coordinate attributes are only one possible source of texture coordinates. Owing to the power of modern fragment shaders, texture coordinates need not come directly from per-vertex attributes. A texture lookup may be evaluated from a set of coordinates generated *in the fragment shader itself* as the result of a computation involving other per-vertex attributes.

A texture coordinate generated in the fragment shader can even be the *result of an earlier texture lookup* in that same fragment shader. In this technique the texture image values in the first texture are not colors, but rather

texture coordinates themselves. This is an extremely powerful technique called *indirect texturing*. The first texture lookup forms a "table lookup," or "indirection," that generates a new texture coordinate for the second texture lookup.

Indirect texturing is an example of a more general case of texturing in which evaluating a texture sample generates a "value" other than a color. Clearly, not all texture lookups are used as colors. However, for ease of understanding in the following discussion, we will assume that the texture image's values represent the most common case—colors.

9.7 Evaluating the Fragment Shader

Armed with the current shader uniform values and interpolated per-vertex attributes at the fragment center, we are ready to compute the fragment's color by evaluating (or "running") the fragment shader. All of the source values are in place. Note, however, that we have not yet evaluated the texture lookups in the shader. Some of the earliest shading languages required that the textures be addressed only by per-vertex attributes, and in some cases, actually computed the texture lookups before even invoking the fragment shader. However, as discussed above, modern shaders allow for texture coordinates to be computed in the fragment shader itself, perhaps even as the result of a texture lookup. Also, conditionals and varying loop iterations in a shader may cause texture lookups to be skipped for some fragments. As a result, we will consider the rasterization of textures to be a part of the fragment shader itself.

In fact, while the mathematical computations that are done inside of the fragment shader are interesting, the most (mathematically) complex part of an isolated fragment shader evaluation is the computation of the texture lookups. The texture lookups are, as we shall see, far more than merely grabbing and returning the closest texel to the fragment center. The wide range of mappings of textures onto geometry and then geometry into fragments requires a much larger set of techniques to avoid glaring visual artifacts. The next section will describe in detail these complexities.

9.8 Rasterizing Textures

The previous section described how to interpolate per-vertex texture coordinate-attributes for use in a fragment shader, but this is only the first step in evaluating a texture lookup in a fragment shader. Having computed or interpolated the texture coordinate for a given fragment, the texture coordinate must be mapped into the texture image itself to produce a color.

9.8.1 Texture Coordinate Review

We will be using a number of different forms of coordinates throughout our discussion of rasterizing textures. This includes the application-level, normalized, texel-independent *texture coordinates* (u, v), as well as the texture size-dependent texel coordinates (u_{texel}, v_{texel}), both of which are considered real values. We used these coordinates in our introduction to texturing.

A final form of texture coordinate is the *integer texel coordinate*, or *texel address*. These represent direct indexing into the texture image array. Unlike the other two forms of coordinates, these are (as the name implies) integral values. The mapping from texel coordinates to integer texel coordinates is not universal and is dependent upon the texture filtering mode, which will be discussed below.

9.8.2 Mapping a Coordinate to a Texel

When rasterizing textures, we will find that — due to the nature of perspective projection, the shape of geometric objects, and the way texture coordinates are generated — fragments will rarely correspond directly and exactly to texels in a one-to-one mapping. Any rasterizer that supports texturing needs to handle a wide range of texel-to-fragment mappings. In the initial discussions of texturing in Chapter 7, we noted that texel coordinates generally include precision (via either floating-point or fixed-point numbers) that is much more fine-grained than the per-texel values that would seem to be required. As we shall see, in several cases we will use this so-called subtexel precision to improve the quality of rendered images in a process known as *texture filtering*.

Texture filtering (in its numerous forms) performs the mapping from real-valued texel coordinates to final texture image values or colors through a mixture of texel coordinate mapping and combinations of the values of the resulting texel or texels. We will break down our discussion of texture filtering into two major cases: one in which a single texel maps to an area that is the size of multiple fragments (magnification), and one in which a number of texels map into an area covered by a single fragment (minification), as they are handled quite differently.

Magnifying a Texture

Our initial texturing discussion stated that one common method of mapping these subtexel precise coordinates to texture image colors was simply to select

the texel containing the fragment center point and use its color directly. This method, called *nearest-neighbor texturing*, is very simple to compute. For any (u_{texel}, v_{texel}) texel coordinate, the integer texel coordinate (u_{int}, v_{int}) is the nearest integer texel center, computed via rounding:

$$(u_{int}, v_{int}) = (\lfloor u_{texel} + 0.5 \rfloor, \lfloor v_{texel} + 0.5 \rfloor)$$

Having computed this integer texel coordinate, we simply use the *Image*() function to look up the value of the texel. The returned color is passed to the fragment shader for the current fragment. While this method is easy and fast to compute, it has a significant drawback when the texture is mapped in such a way that a single texel covers more than one pixel. In such a case, the texture is said to be "magnified," as a quadrilateral block of multiple fragments on the screen is entirely covered by a single texel in the texture, as can be seen in Figure 9.12.

FIGURE 9.12 Nearest-neighbor magnification.

With nearest-neighbor texturing, all (u_{texel}, v_{texel}) texel coordinates in the square

$$i_{int} - 0.5 \leq u_{texel} < i_{int} + 0.5$$

$$j_{int} - 0.5 \leq v_{texel} < j_{int} + 0.5$$

will map to the integer texel coordinates (i_{int}, j_{int}) and thus produce a constant fragment shader value. This is a square of height and width 1 in texel space, centered at the texel center. This results in obvious squares of constant color, which tends to draw attention to the fact that a low-resolution image has been mapped onto the surface. See Figure 9.12 for an example of a nearest-neighbor filtered texture used with a fragment shader that returns the texture as the final output color directly. In most cases, this blocky result is not the desired visual impression.

The problem lies with the fact that nearest-neighbor texturing represents the texture image as a piecewise constant function of (u, v). The resulting fragment shader attribute is constant across all fragments in a triangle until either u_{int} or v_{int} changes. Since the *floor* operation is discontinuous at integer values, this leads to sharp edges in the function represented by the texture over the surface of the triangle.

The common solution to the issue of discontinuous colors at texel boundaries is to treat the texture image values as specifying a different kind of function. Rather than creating a piecewise constant function from the discrete texture image values, we create a piecewise smooth color function. While there are many ways to create a smooth function from a set of discrete values, the most common method in rasterization hardware is linearly interpolating between the colors at each texel center in two dimensions. The method first computes the maximum integer texel coordinate (u_{int}, v_{int}) that is less than (u_{texel}, v_{texel}), the texel coordinate (i.e., the floor of the texel coordinates):

$$(u_{int}, v_{int}) = (\lfloor u_{texel} \rfloor, \lfloor v_{texel} \rfloor)$$

In other words, (u_{int}, v_{int}) defines the minimum (lower left in texture image space) corner of a square of four adjacent texels that "bound" the texel coordinate (Figure 9.13). Having found this square, we can also compute a *fractional texel coordinate* $0.0 \leq u_{frac}, v_{frac} < 1.0$ that defines the position of the texel coordinate within the 4-texel square (Figure 9.14).

$$(u_{frac}, v_{frac}) = (u_{texel} - u_{int}, v_{texel} - v_{int})$$

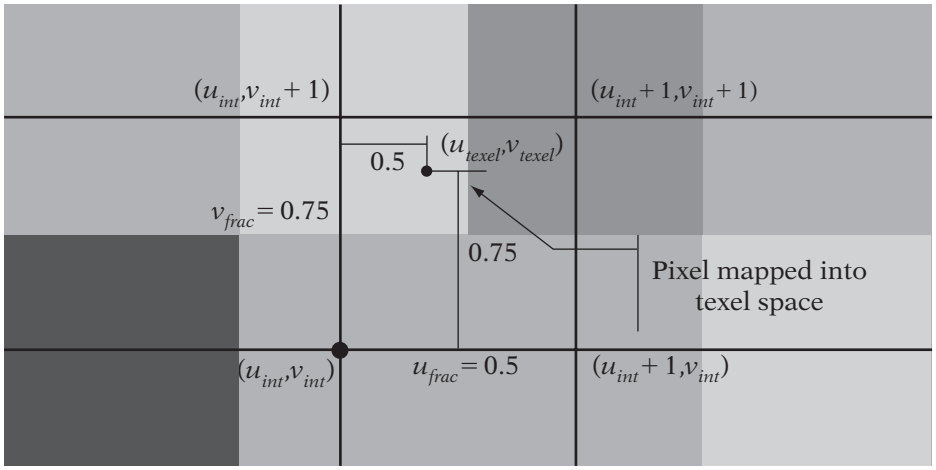


FIGURE 9.13 Finding the four texels that "bound" a pixel center and the fractional position of the pixel.

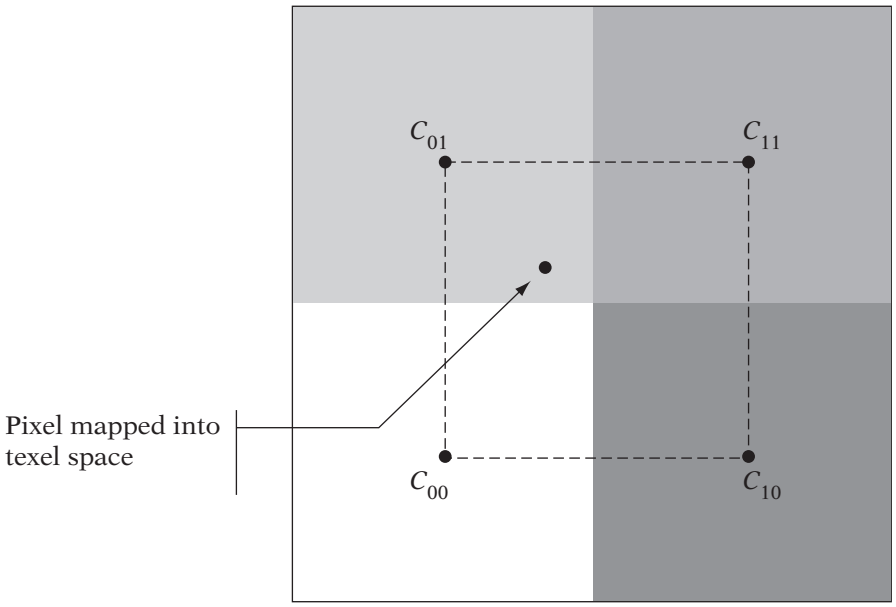


FIGURE 9.14 The four corners of the texel-space bounding square around the pixel center.

We use *Image*() to look up the texel colors at the four corners of the square. For ease of notation, we define the following shorthand for the color of the texture at each of the four corners of the square (Figure 9.14):

$$C_{00} = Image(u_{int}, v_{int})$$
$$C_{10} = Image(u_{int} + 1, v_{int})$$
$$C_{01} = Image(u_{int}, v_{int} + 1)$$
$$C_{11} = Image(u_{int} + 1, v_{int} + 1)$$

Then, we define a smooth interpolation of the four texels surrounding the texel coordinate. We define the smooth mapping in two stages, as shown in Figure 9.15. First, we linearly interpolate between the colors along the minimum-v edge of the square, based on the fractional u coordinate:

$$C_{MinV} = C_{00}(1 - u_{frac}) + C_{10}u_{frac}$$

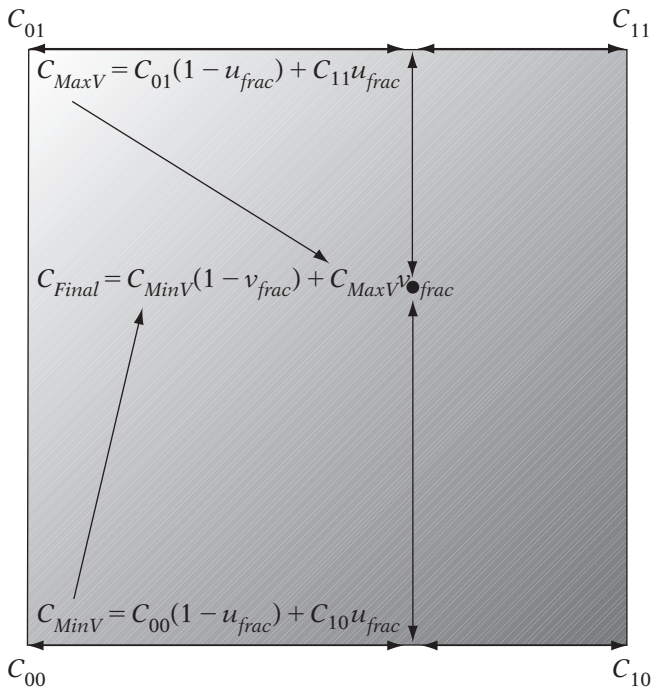


FIGURE 9.15 Bilinear filtering.

and similarly along the maximum-v edge:

$$C_{MaxV} = C_{01}(1 - u_{frac}) + C_{11}u_{frac}$$

Finally, we linearly interpolate between these two values using the fractional v coordinate:

$$C_{Final} = C_{MinV}(1 - v_{frac}) + C_{MaxV}v_{frac}$$

See Figure 9.15 for a graphical representation of these two steps. Substituting these into a single, direct formula, we get

$$\begin{aligned} C_{Final} = {} & C_{00}(1 - u_{frac})(1 - v_{frac}) + C_{10}u_{frac}(1 - v_{frac}) \\ & + C_{01}(1 - u_{frac})v_{frac} + C_{11}u_{frac}v_{frac} \end{aligned}$$

This is known as *bilinear texture filtering* because the interpolation involves linear interpolation in two dimensions to generate a smooth function from four neighboring texture image values. It is extremely popular in hardware 3D graphics systems. The fact that we interpolated along u first and then interpolated along v does not affect the result (other than by potential precision issues). A quick substitution shows that the results are the same either way. However, note that this is *not* an affine mapping. Affine mappings in 2D are uniquely defined by three distinct points. The fourth source point of our bilinear texture mapping may not fit the mapping defined by the other three points.

Using bilinear filtering, the colors across the entire texture domain are continuous. An example of the visual difference between nearest-neighbor and bilinear filtering is shown in Figure 9.16. While bilinear filtering can

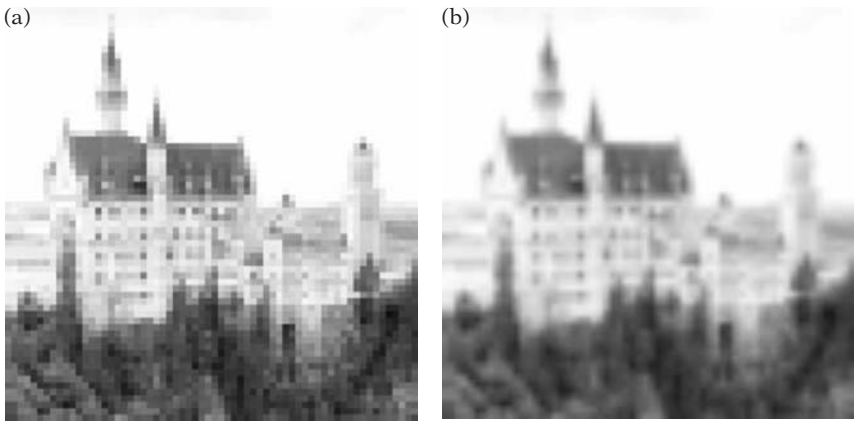


FIGURE 9.16 Extreme magnification of a texture (a) using nearest-neighbor filtering and (b) using bilinear filtering.

greatly improve the image quality of magnified textures by reducing the visual "blockiness," it will not add new detail to a texture. If a texture is magnified considerably (i.e., one texel maps to many pixels), the image will look blurry due to this lack of detail. The texture shown in Figure 9.16 is highly magnified, leading to obvious blockiness in the left image (a) and blurriness in the right image (b).

Texture Magnification in Practice

The `Iv` APIs use the `IvTexture` function `SetMagFiltering` to control texture magnification. `Iv` supports both bilinear filtering and nearest-neighbor selection. They are each set as follows:

```
IvTexture* texture;

// ...

{
   // Nearest-neighbor
   texture->SetMagFiltering(kNearestTexMagFilter);

   // Bilinear interpolation
   texture->SetMagFiltering(kBilerpTexMagFilter);

   // ...
```

Minifying a Texture

Throughout the course of our discussions of rasterization so far, we have mainly referred to fragments by their centers — infinitesimal points located at the center of a square fragment (continuing to assume only complete fragments for now). However, fragments have nonzero area. This difference between the area of a fragment and the point sample representing it becomes very obvious in a common case of texturing.

As an example, imagine an object that is distant from the camera. Objects in a scene are generally textured at high detail. This is done to avoid the blurriness (such as the blurriness we saw in Figure 9.16(b)) that can occur when an object that is close to the camera has a low-resolution texture applied to it. As that same object and texture is moved into the distance (a common situation in a dynamic scene), this same, detailed texture will be mapped to smaller and smaller regions of the screen due to perspective scaling of the object. This is known as *minification* of a texture, as it is the inverse of magnification. This results in the same object and texture covering fewer and fewer fragments.

In an extreme (but actually quite common) case, the entire high-detail texture could be mapped in such a way that it maps to only a few fragments. Figure 9.17 provides such an example; in this case, note that if the object moves even slightly (even less than a pixel), the exact texel covering the fragment's center point can change drastically. In fact, such a point sample is almost random in the texture and can lead to the point-sampled color of the texture used for the fragment changing wildly from frame to frame as the object moves in tiny, subpixel amounts on the screen. This can lead to flickering over time, a distracting artifact in an animated, rendered image.

The problem lies in the fact that most of the texels in the texture have an almost equal "claim" to the fragment, as all of them are projected within the rectangular area of the fragment. The overall color of the fragment's texture sample *should* represent all of the texels that fall inside of it. One way of thinking of this is to map the square of a complete fragment on the projection plane onto the plane of the triangle, giving a (possibly skewed) quadrilateral, as seen in Figure 9.18. In order to evaluate the color of the texture for that

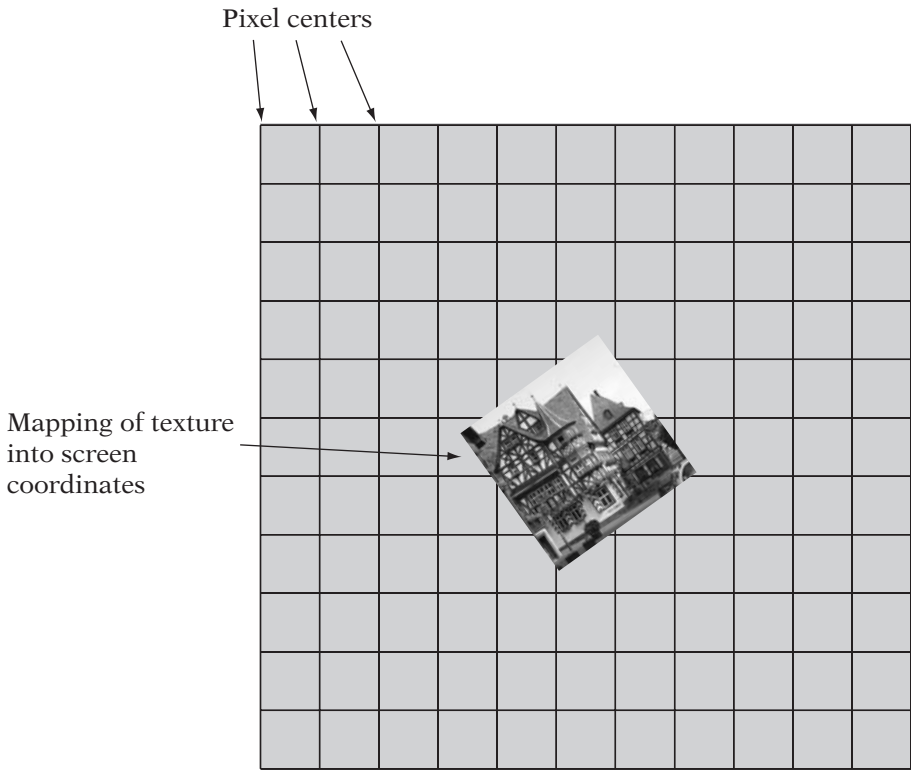


FIGURE 9.17 Extreme minification of a texture.

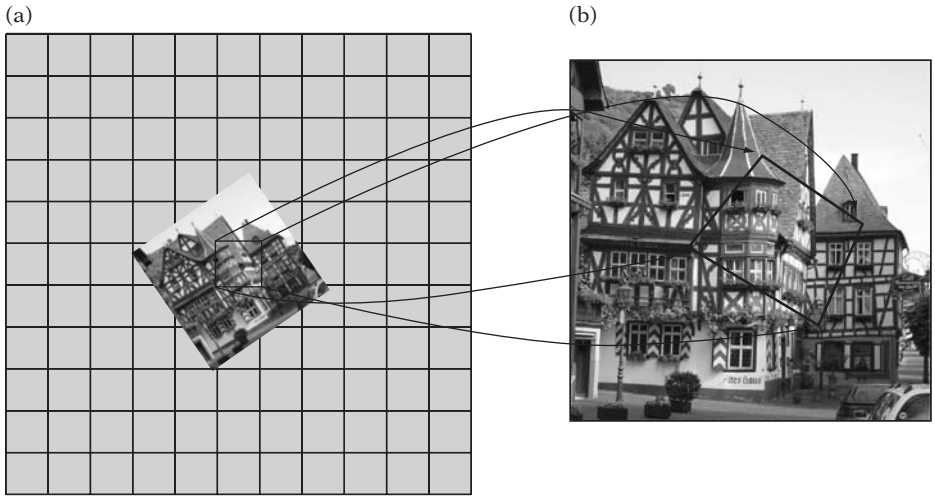


FIGURE 9.18 Mapping the square screen-space area of a pixel back into texel space: (a) screen space with pixel of interest highlighted and (b) texel-space back-projection of pixel area.

fragment fairly, we need to compute a weighted average of the colors of all of the texels in this quadrilateral, based on the relative area of the quadrilateral covered by each texel. The more of the fragment that is covered by a given texel, the greater the contribution of that texel's color to the final color of the fragment's texture sample.

While an exact area-weighted-average method would give a correct fragment color and would avoid the issues seen with point sampling, in reality this is not an algorithm that is best suited for real-time rasterization. Depending on how the texture is mapped, a fragment could cover an almost unbounded number of texels. Finding and summing these texels on a per-fragment basis would require a potentially unbounded amount of per-fragment computation, which is well beyond the means of even hardware rasterization systems. A faster (preferably constant-time) method of approximating this texel averaging algorithm is required. For most modern graphics systems, a method known as *mipmapping* satisfies these requirements.

9.8.3 MIPMAPPING

SOURCE CODE DEMO Mipmapping

Mipmapping [120] is a texture-filtering method that avoids the per-fragment expense of computing the average of a large number of texels. It does so by precomputing and storing additional information with each texture, requiring some additional memory over standard texturing. Mipmapping is a constant-time operation per texture sample and requires a fixed amount

of extra storage per texture (in fact, it increases the number of texels that must be stored by approximately one-third). Mipmapping is a popular filtering algorithm in both hardware and software rasterizers and is relatively simple conceptually.

To understand the basic concept behind mipmapping, imagine a 2×2–texel texture. If we look at a case where the entire texture is mapped to a single fragment, we could replace the 2×2 texture with a 1×1 texture (a single color). The appropriate color would be the mean of the four texels in the 2×2 texture. We could use this new texture directly. If we precompute the 1×1–texel texture at load-time of our application, we can simply choose between the two textures as needed (Figure 9.19).

When the given fragment maps in such a way that it only covers one of the four texels in the original 2×2–texel texture, we simply use a magnification method and the original 2×2 texture to determine the color. If the fragment covers the entire texture, we would use the 1×1 texture directly, again applying the magnification algorithm to it (although with a 1×1 texture, this is just the single texel color). The 1×1 texture adequately represents the overall color of the 2×2 texture in a single texel, but it does not include the detail of the

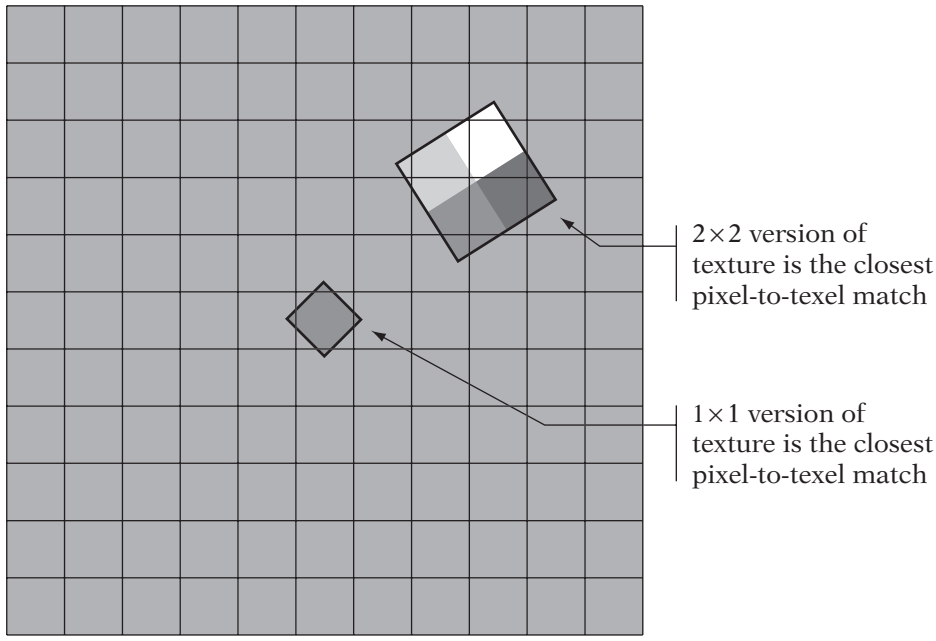

2×2 version of texture is the closest pixel-to-texel match
1×1 version of texture is the closest pixel-to-texel match
Screen-space geometry
(same mipmapped texture applied to both squares)

FIGURE 9.19 Choosing between two sizes of a texture.

original 2×2 texel texture. Each of these two versions of the texture has a useful feature that the other does not.

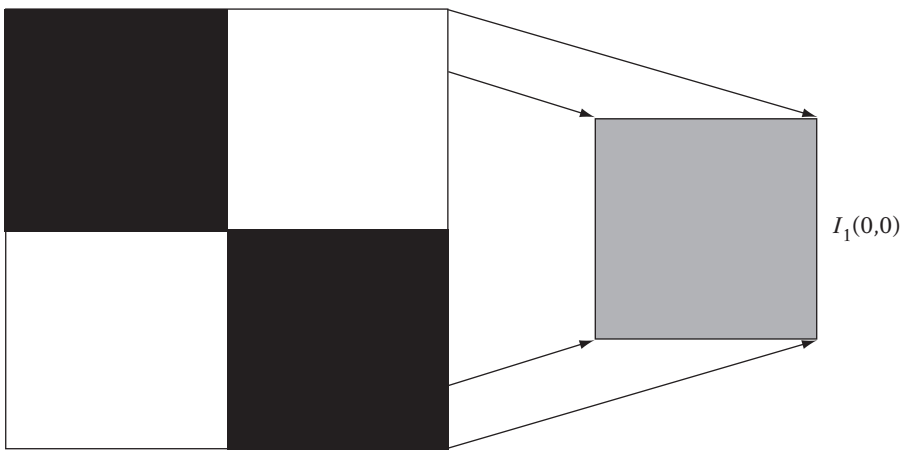
Mipmapping takes this method and generalizes it to any texture with power-of-two dimensions. For the purposes of this discussion, we assume that textures are square (the algorithm does not require this, as we shall see later in our discussion of mipmapping in practice). Mipmapping takes the initial texture image $Image_0$ (abbreviated I_0) of dimension $w_{texture} = h_{texture} = 2^L$ and generates a new version of the texture by averaging each square of four adjacent texels into a single texel. This generates a texture image $Image_1$ of size

$$\frac{1}{2}w_{texture} = \frac{1}{2}h_{texture} = 2^{L-1}$$

as follows:

$$Image_1(i, j) = \frac{I_0(2i, 2j) + I_0(2i+1, 2j) + I_0(2i, 2j+1) + I_0(2i+1, 2j+1)}{4}$$

where $0 \leq i, j < \frac{1}{2}w_{texture}$. Each of the texels in $Image_1$ represents the overall color of a block of the corresponding four texels in $Image_0$ (Figure 9.20).



$$I_1(0,0) = \frac{I_0(0,0) + I_0(1,0) + I_0(0,1) + I_0(1,1)}{4}$$

$$I_1(0,0) = \frac{(1,1,1) + (0,0,0) + (0,0,0) + (1,1,1)}{4} = (\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2})$$

FIGURE 9.20 Texel block to texel mapping between mipmap levels.

Note that if we use the same original texture coordinates for both versions of the texture, $Image_1$ simply appears as a blurry version of $Image_0$ (with half the detail of $Image_0$). If a block of about four adjacent texels in $Image_0$ covers a fragment, then we can simply use $Image_1$ when texturing. But what about more extreme cases of minification? The algorithm can be continued recursively. For each image $Image_i$ whose dimensions are greater than 1, we can define $Image_{i+1}$, whose dimensions are half of $Image_i$, and average texels of $Image_i$ into $Image_{i+1}$. This generates an entire set of $L + 1$ versions of the original texture, where the dimensions of $Image_i$ are equal to

$$\frac{w_{texture}}{2^i}$$

This forms a pyramid of images, each one-half the dimensions (and containing one-quarter the texels) of the previous image in the pyramid. Figure 9.21 provides an example of such a pyramid. We compute this pyramid for each texture in our scene once at load-time or as an offline preprocess and store each entire pyramid in memory. This simple method of computing the mipmap images is known as *box filtering* (as we are averaging a 2×2 "box" of texels into a single texel). Box filtering is not the sole method for generating the mipmap pyramid, nor is it the highest quality. Other, more complex methods are often used to filter each mipmap level down to the next lower level. These methods can avoid some of the visual issues that can crop up from the simple box filter. See Foley et al. [38] and Wohlberg [122] for details of other image-filtering methods.

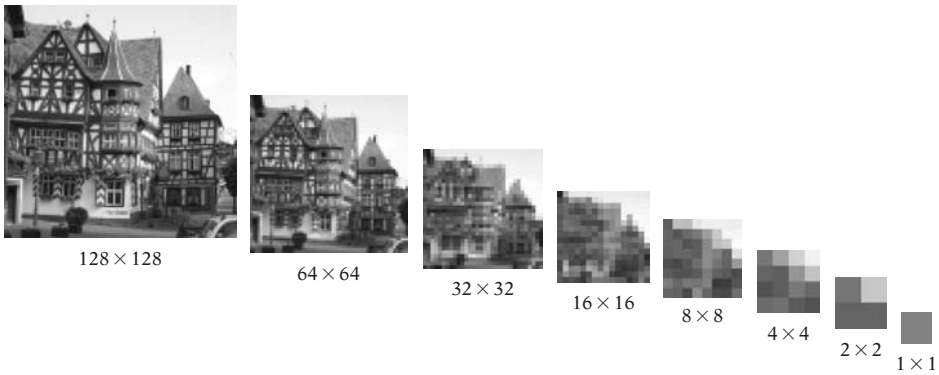


FIGURE 9.21 Mipmap level size progression.

Texturing a Fragment with a Mipmap

The most simple, general algorithm for texturing a fragment with a mipmap can be summarized as follows:

1. Determine the mapping of the fragment in screen space back into a quadrilateral in texture space by determining the texture coordinates at the corners of the fragment.
2. Having mapped the fragment square into a quadrilateral in texture space, select whichever mipmap level comes closest to exactly mapping the quadrilateral to a single texel.
3. Texture the fragment with the "best match" mipmap level selected in the previous step, using the desired magnification algorithm.

There are numerous common ways of determining the "best match" mipmap level, and there are numerous methods of filtering this mipmap level into a final fragment texture value. We would like to avoid having to explicitly map the fragment's corners back into texture space, as this is expensive to compute. We can take advantage of information that other rasterization stages already need. As a part of rasterization, it is common to compute the difference between the texel coordinates at a given fragment center and those of the fragment to the right and below the given fragment. Such differences are used to step the texture coordinates from one fragment to the adjacent fragment, one pixel away. These differences are written as derivatives. The listing that follows is designed to assign intuitive values to each of these four partial derivatives. For those unfamiliar with ∂, it is the symbol for a partial derivative, a basic concept of multivariable calculus. The ∂ operator represents how much one component of the output of a vector-valued function changes when you change one of the input components.

$$\frac{\partial u_{texel}}{\partial x_s} = \text{Change in } u_{texel} \text{ per horizontal pixel step}$$

$$\frac{\partial u_{texel}}{\partial y_s} = \text{Change in } u_{texel} \text{ per vertical pixel step}$$

$$\frac{\partial v_{texel}}{\partial x_s} = \text{Change in } v_{texel} \text{ per horizontal pixel step}$$

$$\frac{\partial v_{texel}}{\partial y_s} = \text{Change in } v_{texel} \text{ per vertical pixel step}$$

If a fragment maps to about one texel, then

$$\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2 \approx 1, \text{ and } \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2 \approx 1$$

In other words, even if the texture is rotated, if the fragment is about the same size as the texel mapped to it, then the overall change in texture coordinates over a single fragment has a length of about one texel. Note that all four of these differences are independent. These partials are dependent upon u_{texel} and v_{texel}, which are in turn dependent upon texture size. In fact, for each of these differentials, moving from $Image_i$ to $Image_{i+1}$ causes the differential to be halved. As we shall see, this is a useful property when computing mipmapping values.

A common formula that is used to turn these differentials into a metric of pixel-texel size ratio is described in Heckbert [55], which defines a formula for the *radius* of a pixel as mapped back into texture space. Note that this is actually the maximum of two radii, the radius of the pixel in u_{texel} and the radius in v_{texel}:

$$size = \max\left(\sqrt{\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2}, \sqrt{\left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2}\right)$$

We can see (by substituting for the ∂) that this value is halved each time we move from $Image_i$ to $Image_{i+1}$ (as all of the ∂ values will halve). So, in order to find a mipmap level at which we map one texel to the complete fragment, we must compute the L such that

$$\frac{size}{2^L} \approx 1$$

where $size$ is computed using the texel coordinates for $Image_0$. Solving for L,

$$L = \log_2 size$$

This value of L is the mipmap level index we should use. Note that if we plug in partials that correspond to an exact one-to-one texture-to-screen mapping,

$$\frac{\partial u_{texel}}{\partial x_s} = 1, \frac{\partial v_{texel}}{\partial x_s} = 0, \frac{\partial u_{texel}}{\partial y_s} = 0, \frac{\partial v_{texel}}{\partial y_s} = 1$$

we get $size = 1$, which leads to $L = 0$, which corresponds to the original texture image as expected.

This gives us a closed-form method that can convert existing partials (used to interpolate the texture coordinates across a scan line) to a specific mipmap level L. The final formula is

$$
\begin{aligned}
L &= \log_2\left(\max\left(\sqrt{\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2}, \sqrt{\left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2}\right)\right) \\
&= \log_2\left(\sqrt{\max\left(\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2, \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2\right)}\right) \\
&= \frac{1}{2}\log_2\left(\max\left(\left(\frac{\partial u_{texel}}{\partial x_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial x_s}\right)^2, \left(\frac{\partial u_{texel}}{\partial y_s}\right)^2 + \left(\frac{\partial v_{texel}}{\partial y_s}\right)^2\right)\right)
\end{aligned}
$$

Note that the value of L is real, not integer (we will discuss the methods of mapping this value into a discrete mipmap pyramid later). The preceding function is only one possible option for computing the mipmap level L. Graphics systems use numerous simplifications and approximations of this value (which is itself an approximation) or even other functions to determine the correct mipmap level. In fact, the particular approximations of L used by some hardware devices are so distinct that some experienced users of 3D hardware can actually recognize a particular piece of display hardware by looking at rendered, mipmapped images. Other pieces of 3D hardware allow the developer (or even the end user) to bias the L values used, as some users prefer "crisp" images (biasing L in the negative direction, selecting a larger, more detailed mipmap level and more texels per fragment) while others prefer "smooth" images (biasing L in the positive direction, tending toward a less detailed mipmap level and fewer texels per fragment). For a detailed derivation of one case of mipmap level selection, see page 106 of Eberly [25].

Another method that has been used to lower the per-fragment expense of mipmapping is to select an L value and thus an single mipmap level per triangle in each frame and rasterize the entire triangle using that mipmap level. While this method does not require any per-fragment calculations of L, it can lead to serious visual artifacts, especially at the edges of triangles, where the mipmap level may change sharply. Software rasterizers that support mipmapping often use this method, known as *per-triangle mipmapping*.

Note that by its very nature, mipmapping tends to use smaller textures on distant objects. When used with software rasterizers, this means that mipmapping can actually *increase* performance, because the smaller mipmap levels are more likely to fit in the processor's cache than the full-detail texture. Most software rasterizers that support texturing are performance bound to some degree by the memory bandwidth of reading textures. Keeping a texture in the cache can decrease these bandwidth requirements significantly. Furthermore,

if point sampling is used with a nonmipmapped texture, adjacent pixels may require reading widely separated parts of the texture. These large per-pixel strides through a texture can result in horrible cache behavior and can impede the performance of nonmipmapped rasterizers severely. These cache miss stalls make the cost of computing mipmapping information (at least on a per-triangle basis) worthwhile, independent of the significant increase in visual quality. In fact, many hardware platforms also see performance increases when using mipmapping, owing to the small, on-chip texture cache memories used to hold recently accessed texture image regions.

Texture Filtering and Mipmaps

The methods described above work on the concept that there will be a single, "best" mipmap level for a given fragment. However, since each mipmap level is twice the size of the next mipmap level in each dimension, the "closest" mipmap level may not be an exact fragment-to-texel mapping. Rather than selecting a given mipmap level as the best, linear mipmap filtering uses a method similar to (bi)linear texture filtering. Basically, mipmap filtering uses the real-valued L to find the pair of adjacent mipmap levels that bound the given fragment-to-texel ratio, $\lfloor L \rfloor$ and $\lceil L \rceil$. The remaining fractional component $(L - \lfloor L \rfloor)$ is used to blend between texture colors found in the two mipmap levels.

Put together, there are now two independent filtering axes, each with two possible filtering modes, leading to four possible mipmap filtering modes as shown in Table 9.1. Of these methods, the most popular is *linear-bilinear*, which is also known as *trilinear interpolation* filtering, or *trilerp*, as it is the

TABLE 9.1 Mipmap filtering modes

Mipmap Filter	***Texture Filter***	***Result***
Nearest	Nearest	Select "best" mipmap level and then select closest texel from it
Nearest	Bilinear	Select "best" mipmap level and then interpolate four texels from it
Linear	Nearest	Select two "bounding" mipmap levels, select closest texel in each, and then interpolate between the two texels
Linear	Bilinear	Select two "bounding" mipmap levels, interpolate four texels from each, and then interpolate between the two results; also called trilerp

exact 3D analog to bilinear interpolation. It is the most expensive of these mipmap filtering operations, requiring the lookup of eight texels per fragment, as well as seven linear interpolations (three per each of the two mipmap levels, and one additional to interpolate between the levels), but it also produces the smoothest results. Filtering between mipmap levels also increases the amount of texture memory bandwidth used, as the two mipmap levels must be accessed per sample. Thus, multilevel mipmap filtering often counteracts the aforementioned performance benefits of mipmapping on hardware graphics devices.

A final, newer form of mipmap filtering is known as *anisotropic* filtering. The mipmap filtering methods discussed thus far implicitly assume that the pixel, when mapped into texture space, produces a quadrilateral that is fit quite closely by some circle. In other words, cases in which the quadrilateral in texture space is basically square. In practice, this is generally not the case. With polygons in extreme perspective, a complete fragment often maps to a very long, thin quadrilateral in texture space. The standard *isotropic* filtering modes can tend to look too blurry (having selected the mipmap level based on the long axis of the quad) or too sharp (having selected the mipmap level based on the short axis of the quad). Anisotropic texture filtering takes the aspect ratio of the texture-space quadrilateral into account when sampling the mipmap and is capable of filtering nonsquare regions in the mipmap to generate a result that accurately represents the tilted polygon's texturing.

Mipmapping in Practice

The individual levels of a mipmap pyramid may be specified manually in the `Iv` interfaces through the use of the `IvTexture` functions `BeginLoadData` and `EndLoadData`. These functions were briefly described in the introduction to texturing (Chapter 7). However, in the case of mipmaps, we use the argument to these functions, `unsigned int level` (previously defaulted to 0), which specifies the mipmap level. The mipmap level of the highest resolution image is 0. Each subsequent level number (1, 2, 3 . . .) represents the mipmap pyramid image with half the dimensions of the previous level. Some APIs (such as OpenGL) require that a "full" pyramid (all the way down to a 1×1 texel) be specified for mipmapping to work correctly. In practice, it is a good idea to provide a full pyramid for all mipmapped textures. The number of mipmap levels in a full pyramid is equal to

$$Levels = \log_2(max(w_{texture}, h_{texture})) + 1$$

Note that the number of mipmap levels is based on the larger dimension of the texture. Once a dimension falls to one texel, it stays at one texel while

TABLE 9.2 Mipmap level size progression

Level	*Width*	*Height*
0	32	8
1	16	4
2	8	2
3	4	1
4	2	1
5	1	1

the larger dimension continues to decrease. So, for a 32×8–texel texture, the mipmap levels are shown in Table 9.2.

Note that the texels of the mipmap level images set in the array returned by `BeginLoadData` must be computed by the application. `Iv` simply accepts these images as the mipmap levels and uses them directly. Once all of the mipmap levels for a texture are specified, the texture may be used for mipmapped rendering by attaching the texture sampler as a shader uniform. An example of specifying an entire pyramid follows.

```
IvTexture* texture;

// ...

{

   for (unsigned int level = 0;
        level < texture->GetLevels();
        level++) {
      unsigned int width = texture->GetWidth(level);
      unsigned int height = texture->GetHeight(level);
      IvTexColorRGBA* texels
         = (IvTexColorRGBA*)texture->BeginLoadData(level);

      for (unsigned int y = 0; y < height; y++) {
         for (unsigned int x = 0; x < width; x++) {
            IvTexColorRGBA& texel = texels[x + y * width];

            // Set the texel color, based on
            // filtering the previous level...
         }
      }
```

```
        texture->EndLoadData(level);
    }

    // ...
```

As a convenience, APIs such as Iv support automatic box filtering and creation of mipmap pyramids from a single image. In Iv, an application may provide the top-level image via the methods above and then automatically generate the remaining levels via the IvTexture function GenerateMipmapPyramid. The preceding code could be completely replaced with the following automatic mipmap generation.

```
IvTexture* texture;

// ...

{

    unsigned int width = texture->GetWidth();
    unsigned int height = texture->GetHeight();
    IvTexColorRGBA* texels
        = (IvTexColorRGBA*)texture->BeginLoadData();

    for (unsigned int y = 0; y < height; y++) {
        for (unsigned int x = 0; x < width; x++) {
            IvTexColorRGBA& texel = texels[x + y * width];

                // Set the texel color
        }
    }

    texture->EndLoadData(0);

    texture->GenerateMipmapPyramid();
    }

    // ...
```

In order to set the minification filter, the IvTexture function SetMinFiltering is used. Iv supports both nonmipmapped modes (bilinear filtering and nearest-neighbor selection), as well as all four mipmapped modes. The most

common mipmapped mode (as described previously) is trilinear filtering, which is set using

```
IvTexture* texture;

// ...

   texture->SetMinFiltering(kBilerpMipmapLerpTexMinFilter);

   // ...
```

9.9 From Fragments to Pixels

Thus far, this chapter has discussed generating fragments, computing the per-fragment source values for a fragment's shader, and some details of the more complex aspects of evaluating a fragment's shader (texture lookups). However, the first few sections of the chapter outlined the real goal of all of this per-fragment work: to generate the final color of a pixel in a rendered view of a scene. Recall that pixels are the destination values that make up the rectangular gridded screen (or framebuffer). The pixels are "bins" into which we place pieces of surface that impinge upon the area of that pixel. Fragments represent these pixel-sized pieces of surface. In the end, we must take all of the fragments that fall into a given pixel's bin and convert them into a single color and depth for that pixel. We have made two important simplifying assumptions in the chapter so far:

- All fragments are complete; that is, a fragment covers the entire pixel.
- All fragments are opaque; that is, near fragments obscure more distant ones.

Put together, these two assumptions lead to an important overall simplification: the nearest fragment at a given pixel completely determines the color of that pixel. In such a system, all we need do is find the nearest fragment at a pixel, shade that fragment, and write the result to the framebuffer. This was a useful simplifying assumption when discussing visible surface determination and texturing. However, it limits the ability to represent some common types of surface materials. It can also cause jagged visual artifacts at the edges of objects on the screen. As a result, two additional features in modern graphics systems have removed these simplifying assumptions: pixel blending allows fragments to be partially transparent, and antialiasing handles

pixels containing multiple partial fragments. We will close the chapter with a discussion of each.

9.9.1 Pixel Blending

Pixel blending is more commonly referred to by the name of its most ubiquitous special case: alpha blending. Although it is really just a special case of general pixel blending, alpha blending is by far the most common form of pixel blending. It is called alpha blending because it involves interpolating between the existing color at a pixel and the color of a new fragment based on the alpha value (or opacity) of the fragment. However, as we shall see, pixel blending does not always use the alpha channel.

Pixel blending is a per-fragment, nongeometric function that takes as its inputs the shaded color of the current fragment (which we will call C_{src}), the fragment's alpha value (which is properly a component of the fragment color, but which we will refer to as A_{src} for convenience), the current color of the pixel in the framebuffer (C_{dst}), and sometimes an existing alpha value in the framebuffer at that pixel (A_{dst}). These inputs, along with a pair of blending functions F_{src} and F_{dst}, define the resulting color (and potentially alpha value) that will be written to the pixel in the framebuffer, C_P. Note that C_P once written will become C_dest in later blending operations involving the same pixel. The general form of blending is

$$C_P = F_{src}C_{src} + F_{dst}C_{dst}$$

The simplest form of pixel blending is to disable blending entirely ("source replace" mode), in which the fragment replaces the existing pixel. This is equivalent to

$$\begin{aligned} F_{src} &= 1 \\ F_{dst} &= 0 \\ C_P &= F_{src}C_{src} + F_{dst}C_{dst} = (1)C_{src} + (0)C_{dst} = C_{src} \end{aligned}$$

Alpha blending involves using the source alpha value A_{src} as the opacity of the new fragment to linearly interpolate between C_{src} and C_{dst}:

$$\begin{aligned} F_{src} &= A_{src} \\ F_{dst} &= (1 - A_{src}) \\ C_P &= F_{src}C_{src} + F_{dst}C_{dst} = A_{src}C_{src} + (1 - A_{src})C_{dst} \end{aligned}$$

Alpha blending requires C_{dst} as an operand. Because C_{dst} is the pixel color (generally stored in the framebuffer), alpha blending can (depending on the

hardware) require that the pixel color be read from the framebuffer for each fragment blended. This increased memory bandwidth means that alpha blending can impact performance on some systems (in a manner analogous to depth buffering). In addition, alpha blending has several other properties that make its use somewhat challenging in practice.

Alpha blending is designed to compute a new pixel color based on the idea that the new fragment color represents a possibly translucent surface whose opacity is given by A_{src}. Alpha blending only uses the fragment alpha value, not the alpha value of the destination pixel. The existing pixel color is assumed to represent the entirety of the existing scene at that pixel that is more distant than the current fragment, in front of which the translucent fragment is placed. For the following discussion, we will write alpha blending as

$$Blend(C_{src}, A_{src}, C_{dst}) = A_{src}C_{src} + (1 - A_{src})C_{dst}$$

The result of multiple alpha blending operations is order-dependent. Each alpha blending operation assumes that C_{dst} represents the final color of all objects more distant than the new fragment. If we view the blending of two possibly translucent fragments (C_1, A_1) and (C_2, A_2) onto a background color C_0 as a sequence of two blends, we can quickly see that, in general, changing the order of blending changes the result. For example, if we compare the two possible blending orders, set $A_1 = 1.0$, and expand the functions, we get

$$Blend(C_2, A_2, Blend(C_1, A_1, C_0)) \stackrel{?}{=} Blend(C_1, A_1, Blend(C_2, A_2, C_0))$$

$$Blend(C_2, A_2, Blend(C_1, 1.0, C_0)) \stackrel{?}{=} Blend(C_1, 1.0, Blend(C_2, A_2, C_0))$$

$$Blend(C_2, A_2, C_1) \stackrel{?}{=} C_1$$

These two sides are almost never equal; the two blending orders will generally produce different results. In most cases, alpha blending of two surfaces with a background color is order-dependent.

Pixel Blending and Depth Buffering

In practice, this order dependence of alpha blending complicates depth buffering. The depth buffer is based on the assumption that a fragment at a given depth will completely obscure any fragment that is at a greater depth, which is only true for opaque objects. In the presence of alpha blending, we must compute the pixel color in a very specific ordering. We could depth sort all of the triangles, but as discussed above, this is expensive and has serious correctness issues with many datasets. Instead, one option is to use the assumption that for most scenes, the number of translucent triangles is much smaller

than the number of opaque triangles. Given a set of triangles, one method of attempting to correctly compute the blended pixel color is as follows:

1. Collect the opaque triangles in the scene into a list, O.
2. Collect the translucent triangles in the scene into another list, T.
3. Render the triangles in O normally, using depth buffering.
4. Sort the triangles in T by depth into a far-to-near ordering.
5. Render the sorted list T with blending, using depth buffering.

This might seem to solve the problem. However, per-triangle depth sorting is still an expensive operation that has to be done on the host CPU in most cases. Also, per-triangle sorting cannot resolve all differences, as there are common configurations of triangles that cannot be correctly sorted back to front. Other methods have been suggested to avoid both of these issues. One such method is to depth sort at a per-object level to avoid gross-scale out-of-order blending, and then use more complex methods such as depth peeling [34], which uses advanced programmable shading and multiple renderings of objects to "peel away" closer surfaces (using the depth buffer) and generate depth-sorted colors. While quite complicated, the method works entirely on the GPU, and focuses on getting the closest layers correct, under the theory that deeper and deeper layers of transparency gain diminishing returns (as they contribute less and less to the final color).

Depth sorting or depth peeling of pixel-blended triangles can be avoided in some application-specific cases. Two other common pixel blending modes are commutative, and are thus order-independent. The two blending modes are known as *add* and *modulate*. Additive blending creates the effect of "glowing" objects and is defined as follows:

$$
\begin{aligned}
F_{src} &= 1 \\
F_{dst} &= 1 \\
C_P &= F_{src}C_{src} + F_{dst}C_{dst} = (1)C_{src} + (1)C_{dst} = C_{src} + C_{dst}
\end{aligned}
$$

Modulate blending implements color filtering. It is defined as

$$
\begin{aligned}
F_{src} &= 0 \\
F_{dst} &= C_{src} \\
C_P &= F_{src}C_{src} + F_{dst}C_{dst} = (0)C_{src} + C_{src}C_{dst} = C_{src}C_{dst}
\end{aligned}
$$

Note that neither of these effects involves the alpha component of the source or destination color. Both additive and modulate blending modes still

require the opaque objects to be drawn first, followed by the blended objects, but neither requires the blended *objects* to be sorted into a depthwise ordering. As a result, these blending modes are very popular for particle system effects, in which many thousands of tiny, blended triangles are used to simulate smoke, steam, dust, or water.

Note that if depth buffering is used with unsorted, blended objects, the blended objects must be drawn with depth buffer *writing* disabled, or else any out-of-order (front-to-back) rendering of two blended objects will result in the more distant object not being drawn. In a sense, blended objects do not exist in the depth buffer, because they do not obscure other objects.

Blending in Practice

Blending is enabled and controlled quite simply in most graphics systems, although there are many options beyond the modes supported by `Iv`. Enabling and disabling blending and setting the blending mode are done via the `IvRenderer` function `SetBlendFunc`, which sets both F_{src} and F_{dst} in a single function call. To use classic alpha blending, the function call is

```
renderer->SetBlendFunc(kOpacityBlendFunc);
```

Additive mode is set using the call

```
renderer->SetBlendFunc(kAddBlendFunc);
```

Modulate blending may be used via the call

```
renderer->SetBlendFunc(kMultiplyBlendFunc);
```

Blending may be disabled via the call

```
renderer->SetBlendFunc(kNoBlendFunc);
```

This interface is very flexible and direct. There are far more blending functions available in OpenGL (and D3D); these are detailed in the *OpenGL Programming Guide* [85].

Recall that it is often useful to disable z-buffer writing while rendering blended objects. This is accomplished via depth buffer "masking," described previously in the depth buffering section.

9.9.2 Antialiasing

The other simplifying rasterization assumption we made earlier, the idea that partial fragments are either ignored or "promoted" to complete fragments, induces its own set of issues. The idea of converting all fragments into all-or-nothing cases was to allow us to assume that a single fragment would "win" a pixel and determine its color. We used this assumption to reduce per-fragment computations to a single-point sample.

This is reasonable if we treat pixels as pure point samples, with no area. However, in our initial discussion of fragments and our detailed discussion of mipmapped textures, we saw that this is not the case; each pixel represents a rectangular region on the screen with a nonzero area. Because of this, more than one (partial) fragment may be visible inside of a pixel's rectangular region. Figure 9.22 provides an example of such a multifragment pixel.

Using the point-sampled methods discussed, we would select the color of a single fragment to represent the entire area of the pixel. However, as can be seen in Figure 9.23, this pixel center point sample may not represent the color of the pixel as a whole. In the figure, we see that most of the area of the pixel

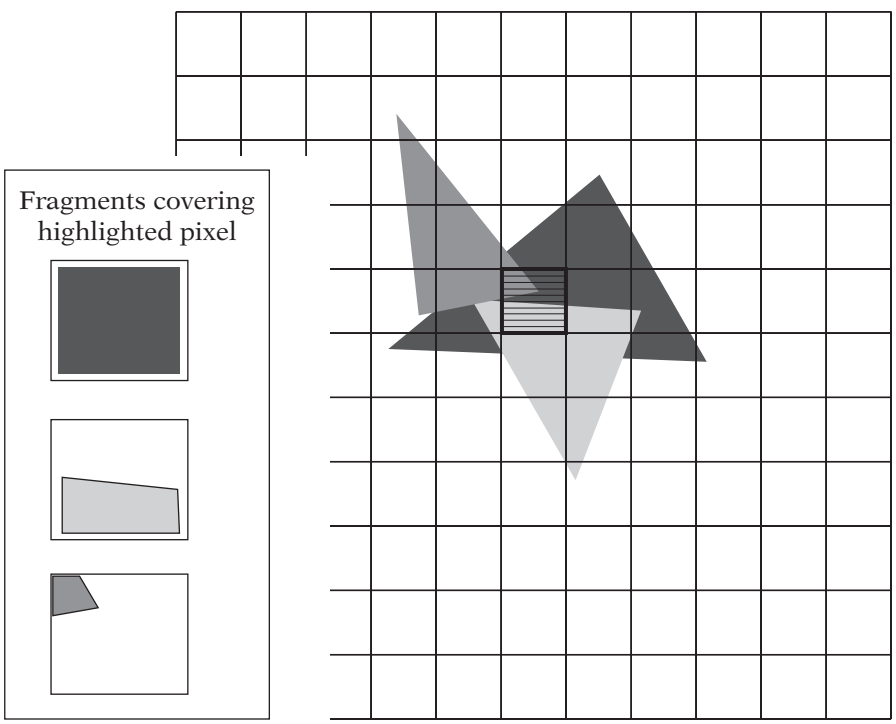


Figure 9.22 Multiple fragments falling inside the area of a single pixel.

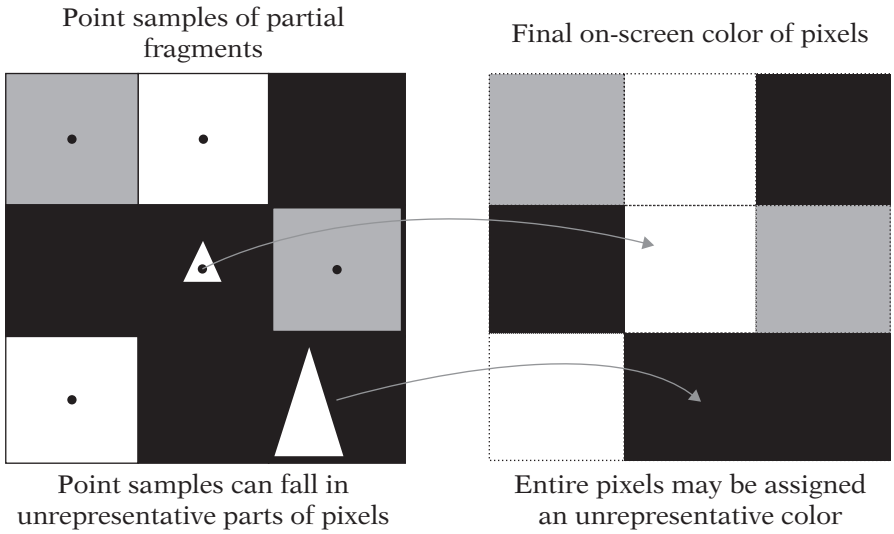

Point samples of partial fragments
Final on-screen color of pixels
Point samples can fall in unrepresentative parts of pixels
Entire pixels may be assigned an unrepresentative color

FIGURE 9.23 A point sample may not accurately represent the overall color of a pixel.

is dark gray, with only a very small square in the center being bright white. As a result, selecting a pixel color of bright white does not accurately represent the color of the pixel rectangle as a whole. Our perception of the color of the rectangle has to do with the relative areas of each color in the rectangle, something that the single point–sampling method cannot represent.

Figure 9.24 makes this even more apparent. In this situation, we see two examples of a pixel of interest (the center pixel in each 9-pixel 3×3 grid). In both center pixel configurations (top and bottom of the left side of the figure), the vast majority of the surface area is dark gray. In each of the two cases, the center pixel contains a small, white fragment. The white fragments are the same size in both cases, but they are in slightly different positions relative to the center pixel in each of the two cases. In the first (top) example, the white fragment happens to contain the pixel center, while in the bottom case, the white fragment does not contain the pixel center. The right column shows the color that will be assigned to the center pixel in each case. Very different colors are assigned to these two pixels, even though their geometric configurations are almost identical. This demonstrates the fact that single-point sampling the color of a pixel can lead to somewhat arbitrary results. In fact, if we imagine that the white fragment were to move across the screen over time, an entire line of pixels would flash between white and gray as the white fragment moved through each pixel's center.

It is possible to determine a more accurate color for the two pixels in the figure. If the graphics system uses the relative areas of each fragment within the pixel's rectangle to weight the color of the pixel, the results will

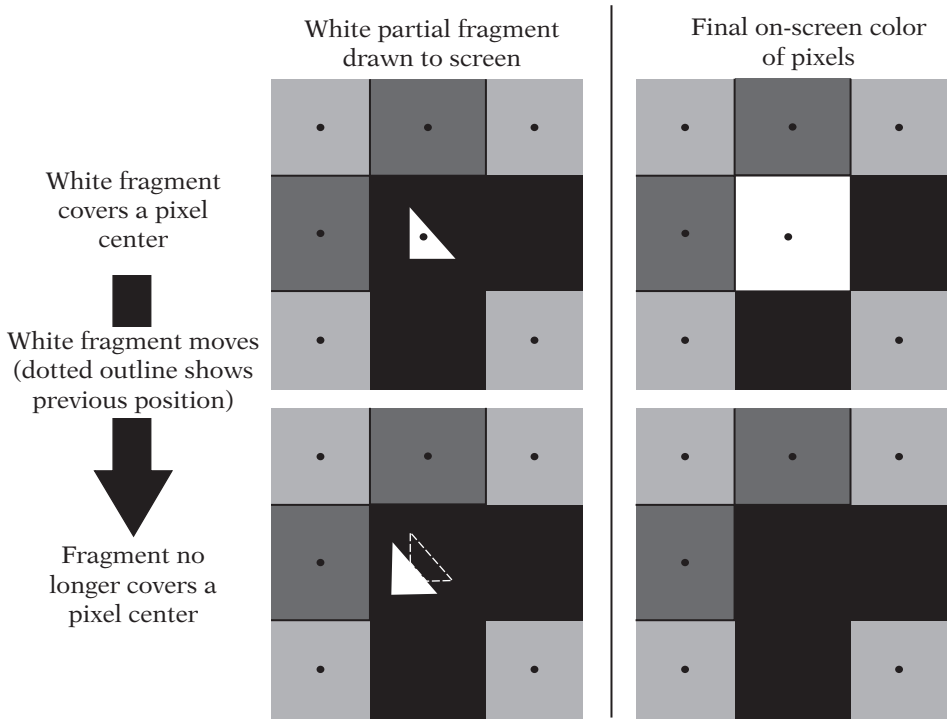

White partial fragment drawn to screen
Final on-screen color of pixels
White fragment covers a pixel center
White fragment moves (dotted outline shows previous position)
Fragment no longer covers a pixel center

FIGURE 9.24 Subpixel motion causing a large change in point-sampled pixel color.

be much better. In Figure 9.25, we can see that the white fragment covers approximately 10 percent of the area of the pixel, leaving the other 90 percent as dark gray. Weighting the color by the relative areas, we get a pixel color of

$$C_{area} = 0.1 \times (1.0, 1.0, 1.0) + 0.9 \times (0.25, 0.25, 0.25) = (0.325, 0.325, 0.325)$$

Note that this computation is independent of *where* the white fragment falls within the pixel; only the size and color of the fragment matter. Such an area-based method avoids the point-sampling errors we have seen. This system can be extended to any number of different colored fragments within a given pixel. Given a pixel with area a_{pixel} and a set of n disjoint fragments, each with an area within the pixel a_i and a color C_i, the final color of the pixel is then

$$\frac{\sum_{i=1}^{n} a_i \times C_i}{a_{pixel}} = \sum_{i=1}^{n} \frac{a_i}{a_{pixel}} \times C_i = \sum_{i=1}^{n} F_i \times C_i$$

where F_i is the fraction of the pixel covered by the given fragment, or the fragment's "coverage." This method is known as *area sampling*. In fact, this

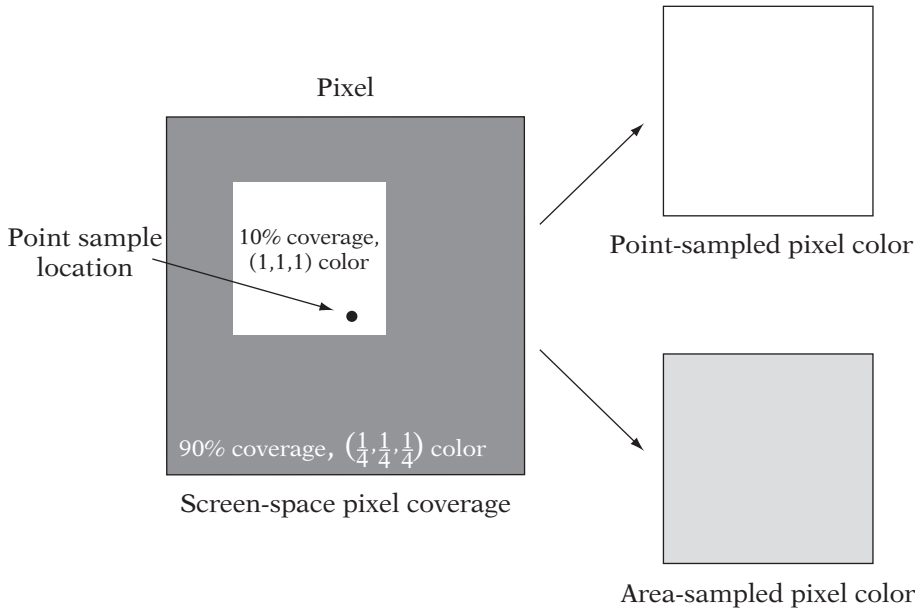


FIGURE 9.25 Area sampling of a pixel.

is really a special case of a more general definite integral. If we imagine that we have a screen-space function that represents the color of every position on the screen (independent of pixels or pixel centers) $C(x, y)$, then the color of a pixel defined as the region $l \leq x \leq r, t \leq y \leq b$ (the left, right, top, and bottom screen coordinates of the pixel), using this area sampling method, is equivalent to

$$\frac{\int_t^b \int_l^r C(x, y)dxdy}{\int_t^b \int_l^r dxdy} = \frac{\int_t^b \int_l^r C(x, y)dxdy}{(b-t)(r-l)} = \frac{\int_t^b \int_l^r C(x, y)dxdy}{a_{pixel}} \tag{9.4}$$

which is the integral of color over the pixel's area, divided by the total area of the pixel. The summation version of equation 9.4 is a simplification of this more general integral, using the assumption that the pixel consists entirely of areas of piecewise constant color, namely, the fragments covering the pixel.

As a verification of this method, we shall assume that the pixel is entirely covered by a single, complete fragment with color $C(x, y) = C_T$, giving

$$\frac{\int_t^b \int_l^r C(x, y)dxdy}{a_{pixel}} = \frac{\int_t^b \int_l^r C_T dxdy}{a_{pixel}} = C_T \frac{\int_t^b \int_l^r dxdy}{a_{pixel}} = C_T \frac{a_{pixel}}{a_{pixel}} = C_T \tag{9.5}$$

which is the color we would expect in this situation.

While area sampling does avoid completely missing or overemphasizing any single sample, it is not the only method used, nor is it the best at representing the realities of display devices (where the intensity of a physical pixel may not actually be constant within the pixel rectangle). The area sampling shown in equation 9.4 implicitly weights all regions of the pixel equally, giving the center of the pixel weighting equal to that of the edges. As a result, it is often called *unweighted area sampling*. *Weighted area sampling*, on the other hand, adds a weighting function that can bias the importance of the colors in any region of the pixel as desired. If we simplify the original pixel boundaries and the functions associated with equation 9.4 such that boundaries of the pixel are $0 \leq x, y \leq 1$, then equation 9.4 becomes

$$\frac{\int_t^b \int_l^r C(x, y)dxdy}{\int_t^b \int_l^r dxdy} = \frac{\int_0^1 \int_0^1 C(x, y)dxdy}{1} \tag{9.6}$$

Having simplified equation 9.4 into equation 9.6, we define a weighting function $W(x, y)$ that allows regions of the pixel to be weighted as desired:

$$\frac{\int_0^1 \int_0^1 W(x, y)C(x, y)dxdy}{\int_0^1 \int_0^1 W(x, y)dxdy} \tag{9.7}$$

In this case, the denominator is designed to normalize according to the weighted area. A similar substitution to equation 9.5 shows that constant colors across a pixel map to the given color. Note also that (unlike unweighted area sampling) the position of a primitive within the pixel now matters. From equation 9.7, we can see that unweighted area sampling is simply a special case of weighted area sampling. With unweighted area sampling, $W(x, y) = 1$, giving

$$\begin{aligned}
&\frac{\int_0^1 \int_0^1 W(x, y)C(x, y)dxdy}{\int_0^1 \int_0^1 W(x, y)dxdy} \\
&= \frac{\int_0^1 \int_0^1 (1)C(x, y)dxdy}{\int_0^1 \int_0^1 (1)dxdy} \\
&= \frac{\int_0^1 \int_0^1 C(x, y)dxdy}{\int_0^1 \int_0^1 dxdy} \\
&= \frac{\int_0^1 \int_0^1 C(x, y)dxdy}{1}
\end{aligned}$$

A full discussion of weighted area sampling, the theory behind it, and numerous common weighting functions is given in Foley et al. [38]. For those

desiring more depth, Glassner [41] and Wohlberg [122] detail a wide range of sampling theory.

Supersampled Antialiasing

The methods so far discussed show theoretical ways for computing area-based pixel colors. These methods require that pixel-coverage values be computed per fragment. Computing analytical (exact) pixel-coverage values for triangles can be complicated and expensive. In practice, the pure area-based methods do not lead directly to simple, fast hardware antialiasing implementations.

The conceptually simplest, most popular antialiasing method is known as oversampling, supersampling, or supersampled antialiasing (SSAA). In SSAA, area-based sampling is approximated by point sampling the scene at more than one point per pixel. In SSAA, fragments are generated not at the per-pixel level, but at the per-sample level. In a sense, SSAA is conceptually little more than rendering the entire scene to a larger (higher-resolution) framebuffer, and then filtering blocks of pixels in the higher-resolution framebuffer down to the resolution of the final framebuffer. For example, the supersampled framebuffer may be N times larger in width and height than the final destination framebuffer on-screen. In this case, every $N \times N$ block of pixels in the supersampled framebuffer will be filtered down to a single pixel in the on-screen framebuffer.

The supersamples are combined into a single pixel color via a weighted (or in some cases unweighted) average. The positions and weights used with weighted area versions of these sampling patterns differ by manufacturer; common examples of sample positions are shown in Figure 9.26. Note that the number of supersamples per pixel varies from as few as 2 to as many as 16. M-sample SSAA represents a pixel as an M-element piecewise-constant function. Partial fragments will only cover some of the point samples in a pixel, and will thus have reduced weighting in the resulting pixel.

Some of the $N \times N$ sample grids also have rotated versions. The reason for this is that horizontal and vertical lines happen with high frequency and are also correlated with the pixel layout itself. By rotating the samples at the correct angle, all N^2 samples are located at distinct horizontal and vertical positions. Thus, a horizontal or vertical edge moving slowly from left to right or top to bottom through a pixel will intersect each sample individually and will thus have a coverage value that changes in $1/N^2$ increments. With screen-aligned $N \times N$ sample patterns, the same moving horizontal and vertical edges would intersect entire rows or columns of samples at once, leading to coverage values that changed in $1/N$ increments. The rotated patterns can take better advantage of the number of available samples.

M-sample SSAA generates M times (as mentioned above, generally 2–16 times) as many fragments per pixel. Each such (smaller) fragment has its own color computed by evaluating per-vertex attributes, texture values, and

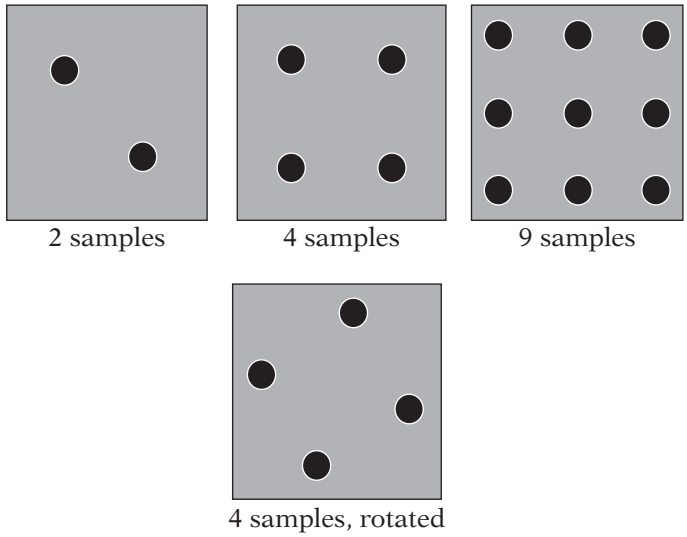


FIGURE 9.26 Common sample-point distributions for multisample-based antialiasing.

the fragment shader itself as many as *M* times more frequently per frame than normal rendering. This per-sample full rendering pipeline is very powerful, since each sample truly represents the color of the geometry at that sample. It is also extremely expensive, requiring the entire rasterization pipeline to be invoked per sample and thus increasing rasterization overhead by 2–16 times. For even powerful 3D hardware systems, this can simply be too expensive.

Multisampled Antialiasing

The most expensive aspect of supersampled antialiasing is the creation of individual fragments per sample and the resulting texturing and fragment shading per sample. Another form of antialiasing recognizes the fact that the most likely causes of aliasing in 3D rendering are partial fragments at the edges of objects, where pixels will contain multiple partial fragments from different objects, often with very different colors. Multisampled antialiasing (MSAA) attempts to fix this issue without raising the cost of rendering as much as does SSAA. MSAA works like normal rendering in that it generates fragments (including partial fragments) at the final pixel size. It only evaluates the fragment shader once per fragment, so the number of fragment shader invocations is reduced significantly when compared to SSAA.

The information that MSAA does add is per-sample fragment coverage. When a fragment is rendered, its color is evaluated once, but then that same

color is stored for each visible sample that the fragment covers. The existing color at a sample (from an earlier fragment) may be replaced with the new fragment's color. But this is done at a per-sample level. At the end of the frame, a "resolve" is still needed to compute the final color of the pixel from the multiple samples. However, only a coverage value (a simple geometric operation) and possibly a depth value is computed per sample, per fragment. The expensive steps of computing a fragment color are still done once per fragment. This greatly reduces the expense of MSAA when compared to SSAA.

There are two subtleties to MSAA worth mentioning. First, since MSAA is coverage-based, no antialiasing is computed on complete fragments. The complete fragment is rendered as if no antialiasing was used. SSAA, on the other hand, antialiases every pixel by invoking the fragment's shader several times per pixel. A key observation is that perhaps the most likely item to cause aliasing in single-sampled complete fragments is texturing (since it is the highest-frequency value across a fragment). Texturing already has a form of antialiasing applied: mipmapping. Thus, this is not a problem for MSAA in most cases.

The other issue is the question of selecting the position in the pixel at which to evaluate a shader on a partial fragment. Normally, we evaluate the fragment shader at the pixel center. However, a partial fragment may not even cover the pixel center. If we sample the fragment shader at the pixel center, we actually will be extrapolating the vertex attributes beyond the intended values. This is particularly noticeable with textures, as we will read the texture at a location that may not have been mapped in the triangle. This can lead to glaring visual artifacts. The solution in most 3D MSAA hardware is to select the centroid of the samples covered by a fragment. Since fragments are convex, the centroid will always fall inside of the fragment. This does add some complexity to the system, but the number of possible configurations of a fragment that does not include the pixel center is limited. The convexity and the fact that the central sample is not touched means that there are a very limited set of covered-sample configurations possible. The set of possible positions can be precalculated before the hardware is even built.

9.9.3 Antialiasing in Practice

For most rendering APIs, the most important step in using MSAA is to create a framebuffer for rendering that is compatible with the technique. Whereas depth buffering required an additional buffer alongside the framebuffer to store the depth values, MSAA requires a special framebuffer format that includes the additional color, depth, and coverage values per sample within each pixel. Different rendering APIs and even different rendering hardware on the same APIs often have different methods for explicitly requesting

MSAA-compatible framebuffers. Some rendering APIs allow the application to specify the number and event layout of samples in the pixel format, while others simply use a single flag for enabling a single (unspecified) level of MSAA. Iv does not support MSAA, so we will describe the methods used in OpenGL and D3D to enable it.

In OpenGL, the creation of the framebuffer is platform-specific. As a result, the specification of MSAA is also platform-specific, often involving vendor-specific extensions. However, the GLUT utility library includes a single flag, `GLUT_MULTISAMPLE`, to be passed to the `glutInitDisplayMode` function. This flag will request that the framebuffer be created with an MSAA-compatible format.

OpenGL also includes a `glEnable/glDisable` flag for MSAA, `GL_MULTISAMPLE`. However, note that many implementations will ignore this flag—if an MSAA-compatible framebuffer is used for rendering, the implementation may simply use MSAA all the time, without regard to this flag.

Finally, some rendering APIs (such as Direct3D) can require special flags or restrictions when presenting an MSAA framebuffer to the screen. In the case of D3D, MSAA framebuffers must be presented to the screen using a special mode that marks the framebuffer's contents as invalid after presentation. This takes into account the fact that the framebuffer must be "resolved" from its multisample-per-pixel format into a single color per pixel during presentation, destroying the multisample information in the process.

9.10 Chapter Summary

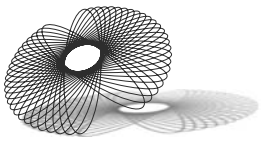
This chapter concludes the discussion of the rendering pipeline. Rasterization provides us with some of the lowest-level yet most mathematically interesting concepts in the entire pipeline. We have discussed the connections between mathematical concepts, such as projective transforms, and rendering methods, such as perspective-correct texturing. In addition, we addressed issues of mathematical precision in our discussion of the depth buffer. Finally, the concept of point sampling versus area sampling appeared twice, relating to both mipmapping and antialiasing. Whether it is implemented in hardware, software, or a mixture of the two, the entire graphics pipeline is ultimately designed only to feed a rasterizer, making the rasterizer one of the most important yet least understood pieces of rendering technology.

Thanks to the availability of high-quality, low-cost 3D hardware on a wide range of platforms, the percentage of readers who will ever have to implement their own rasterizer is now vanishingly small. However, an understanding of how rasterizers function is important even to those who will never need to write one. For example, even a basic practical understanding of the

depth buffering system can help a programmer build a scene that avoids visual artifacts during visible surface determination. Understanding the inner workings of rasterizers can help a 3D programmer quickly debug problems in the geometry pipeline. Finally, this knowledge can guide the programmer to better optimize their geometry pipeline, "feeding" their rasterizer with high-performance datasets.

This page intentionally left blank

Chapter 10
Interpolation



10.1 Introduction

Up to this point, we have considered only motions (more specifically, transformations) that have been created programmatically. In order to create a particular motion (e.g., a submarine moving through the world), we have to write a specific program to generate the appropriate sequence of transformations for our model. However, this takes time and it can be quite tedious to move objects in this fashion. It would be much more convenient to predefine our transformation set in a tool and then somehow regenerate it within our game. An artist could create the sequence using a modeling package, and then a programmer would just write the code to play it back, much as a projector plays back a strip of film. This process of pregenerating a set of data and then playing it back is known as *animation*.

The best way to understand animation is to look at the art form in which it has primarily been used: motion pictures. In this case, the illusion of motion is created by drawing or otherwise recording a series of images on film and then projecting them at 24 or 30 frames per second (for film and video, respectively). The illusion is maintained by a property of the eye–brain combination known as persistence of motion: the eye–brain system sees two frames and invisibly (to our perception) fills in the gaps between them, thus giving us the notion of smooth motion.

We could do something similar in our game. Suppose we had a character that we want to move around the world. The artist could generate various animation sets at 60 frames per second (f.p.s.), and then when we want the character to run, we play the appropriate running animation. When we want the character to walk, we switch to the walking animation. The same process can be used for all the possible motions in the game.

However, there are a number of problems with this. First, by setting the animation set to a rate of 60 f.p.s. and then playing it back directly, we have effectively locked the frame rate for the game at 60 f.p.s. as well. Many monitors can run at 85 f.p.s., and when running in windowed mode, the graphics can be updated much faster than that. It would be much better if we could find some way to generate 85 f.p.s. or more from a 60 f.p.s. dataset. In other words, we need to take our initial dataset and generate a new one at a different rate. This is known as *resampling*.

This brings us to our second problem. Storing 60 f.p.s. per animation adds up to a lot of data. As an example, if we have 10 data points per model that we're storing, with 16 floats per point (i.e., a 4×4 matrix), that adds up to about 38 KB per second of animation. A minute of animation adds up to over 2 MB of data, which can be a serious hit, particularly if we're running on a low-memory platform such as a console. It would be better if we could generate our data at a lower rate, say 10 or 15 f.p.s., and then resample up to the speed we need. This is essentially the same problem as our first one — it's just that our initial dataset has fewer samples.

Alternately, we could take another cue from movie animation. The primary animators on a film draw only the important, infrequent "key" frames that capture the essential flow of an animation. The work of generating the remaining "in-between" frames is left to secondary animators, who generate these intermediate frames from the supplied key frames. These artists are known as 'tweeners. In our case, we could store key frames that store the essential positions of our motion. These key frames would not have to be separated by a constant time interval, rather at smaller intervals when the positions are changing quickly, and at larger intervals when the positions change very slowly. The resampling function would act as our 'tweener for this key frame data.

Fortunately, we have already been introduced to one technique for doing all of this, albeit in another form. This method is known as *interpolation*, and we first saw it when generating a line from two points. Interpolation takes a set of discrete sample points at given time intervals and generates a continuous function that passes through the points. Using this, we can pick any time along the domain of the function and generate a new point so that we might fill in the gaps. We're using the interpolation function to sample at a different rate.

An alternative is approximation, which uses the points to guide the resulting function. In this case, the function does not pass through the points. This may seem odd, but it can help us better control the shape of the function. However, the same principle applies: We generate a function based on the initial sample data and resample later at a different frame rate.

We'll be breaking our discussion of interpolation and approximation into three parts. First, we'll look at some techniques for interpolating and

approximating position. Next, we'll look at how we can extend those techniques for orientation. Finally, we'll look at some applications, in particular, the motion of a constrained camera.

10.2 Interpolation of Position

10.2.1 General Definitions

The general class of functions we'll be using for both interpolating and approximating are called *parametric curves*. We can think of a curve as a squiggle in space, where the parameter controls where we are in the squiggle. The simplest example of a parametric curve is our old line equation,

$$L(t) = P_0 + (P_1 - P_0)t$$

Here t controls where we are on the line, relative to P_0 and P_1.

When curves are used for animation, our parameter is usually represented by u or t. We can think of this as representing time, although the units used don't necessarily have any relationship to seconds. In our discussion we will use u as the parameter to a uniform curve Q such that $Q(0)$ is the start of the curve and $Q(1)$ is the end. When we want to use a general parameterization, we will use t. In this case, we usually set a time value t_i for each point P_i; we expect to end up at position P_i in space at time t_i. The sequence $t_0, t_1, \ldots, t_n$ is sorted (as are the corresponding points) so that it is monotonically increasing.

We can formally define a parametric curve as a function $Q(u)$ that maps an interval of real values (represented by the parameter u, as above) to a continuous set of points. When mapping to $\mathbb{R}^3$, we commonly use a parametric curve broken into three separate functions, one for each coordinate: $Q(u) = (x(u), y(u), z(u))$. This is also known as a *space curve*.

The term *continuous* in our definition is a difficult one to grasp mathematically. Informally, we can think of a continuous function as one that we can draw without ever lifting the pen from the page. Formally, we say that a function f is continuous at a value x_0 if

$$\lim_{x \to x_0} f(x) = f(x_0)$$

In addition, we say that a function $f(x)$ is continuous over an interval (a, b) if it is continuous for every value x in the interval. We can also say that the function has positional, or C^0, continuity over the interval (a, b).

This can be taken further: A function $f(x)$ has tangential, or C^1, continuity across an interval (a, b) if the first derivative $f'(x)$ of the function is continuous across the interval. In our case, the derivative $\mathbf{Q}'(u)$ for parameter u is a tangent vector to the curve at location $Q(u)$. Correspondingly, the derivative of a space curve is $\mathbf{Q}'(u) = (x'(u), y'(u), z'(u))$.

Occasionally, we may be concerned with C^2 continuity, also known as curvature continuity. A function $f(x)$ has C^2 continuity across an interval (a, b) if the second derivative $f''(x)$ of the function is continuous across the interval. Higher orders of continuity are possible, but they are not relevant to the discussion that follows.

A few more definitions will be useful to us. The average speed r we travel along a curve is related to the distance d traveled along the curve and the time it takes to travel that distance, namely,

$$r = d/u$$

The instantaneous speed at a particular parameter u is the length of the derivative vector $\mathbf{Q}'(u)$.

For a given point P on a smooth curve $Q(u)$, we define a circle with first and second derivative vectors equal to those at P as the osculating[1] circle. If the radius of the osculating circle is ρ, the *curvature* κ at P is $1/\rho$. The curvature at any point is always nonnegative. The higher the curvature, the more the curve bends at that point; the curvature of a straight line is 0.

In general, it is not practical to construct a single, closed-form polynomial that uses all of the sample points — most of the curves we will discuss use at most four points as their geometric foundation. Instead, we will create a *piecewise curve*. This consists of curve segments that each apply over a sequential subset of the points and are joined together to create a function across the entire domain. How we create this joint determines the type of continuity we will have in our function as whole. We can achieve C^0 continuity by ensuring that the endpoint of one curve segment is equal to the start point of the next segment. In general, this is desirable.

We can achieve C^1 continuity over the entire piecewise curve by guaranteeing that tangent vectors are equal at the end of one segment and the start of the next segment. A related form of continuity in this case is G^1 continuity, where the tangents at a pair of segment endpoints are not necessarily equal but point in the same direction. In many cases G^1 continuity is good enough for our purposes. And as one might expect, we can achieve C^2 continuity by guaranteeing that the second derivative vectors are equal at the end of one segment and the start of the next segment.

1. So called because it "kisses" up to the point.

10.2.2 Linear Interpolation

Definition

The most basic parametric curve is our example above: a line passing through two points. By using the parameterized line equation based on the two points, we can generate any point along the line. This is known as *linear interpolation* and is the most commonly used form of interpolation in game programming, mainly because it is the fastest. From our familiar line equation,

$$Q(u) = P_0 + u(P_1 - P_0)$$

we can rearrange to get

$$Q(u) = (1 - u)P_0 + uP_1$$

The value u is the factor we use to control our interpolation, or parameter. Recall that if u is 0, $Q(u)$ returns our starting point P_0, and if u is 1, then $Q(u)$ returns P_1, our endpoint. Values of u between 0 and 1 will return a point along the line segment $\overline{P_0P_1}$. When interpolating, we usually care only about values of u within the interval [0, 1] and, in fact, state that the interpolation is undefined outside of this interval.

It is common when creating parametric curves to represent them as matrix equations. As we'll see later, it makes it simple to set certain conditions for a curve and then solve for the equation we want. The standard matrix form is

$$Q(u) = \mathbf{U} \cdot \mathbf{M} \cdot \mathbf{G}$$

where $\mathbf{U}$ is a row matrix containing the polynomial interpolants we're using: $1, u, u^2, u^3$, and so on; $\mathbf{M}$ is a matrix containing the coefficients necessary for the parametric curve; and $\mathbf{G}$ is a matrix containing the coordinates of the geometry that defines the curve. In the case of linear interpolation,

$$\mathbf{U} = \begin{bmatrix} u & 1 \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \end{bmatrix}$$

Note that the columns of $\mathbf{M}$ are the $(u, 1)$ coefficients for P_0 and P_1, respectively.

With this formulation, the result **UMG** will be a 1×3 matrix:

$$\begin{aligned}\mathbf{UMG} &= \begin{bmatrix} x(u) & y(u) & z(u) \end{bmatrix} \\ &= \begin{bmatrix} (1-u)x_0 + ux_1 & (1-u)y_0 + uy_1 & (1-u)z_0 + uz_1 \end{bmatrix}\end{aligned}$$

This is counter to our standard convention of using column vectors. However, rather than write out **G** as individual coordinates, we can write **G** as a column matrix of n points, where for linear interpolation this is

$$\mathbf{G} = \begin{bmatrix} P_0 \\ P_1 \end{bmatrix}$$

Then, using block matrix multiplication, the result **UMG** becomes

$$\mathbf{UMG} = (1-u)P_0 + uP_1$$

This form allows us to use a convenient shorthand to represent a general parameterized curve without having to expand into three essentially similar functions.

Recall that in most cases we are given time values t_0 and t_1 that are associated with points P_0 and P_1, respectively. In other words, we want to start at point P_0 at time t_0 and end up at point P_1 at time t_1. These times are not necessarily 0 and 1, so we'll need to remap our time value t in the interval $[t_0, t_1]$ to a parameter u in the interval $[0, 1]$, which we'll use in our original interpolation equation. If we want the percentage u that a time value t lies between t_0 and t_1, we can use the formula

$$u = \frac{t - t_0}{t_1 - t_0} \tag{10.1}$$

Using this parameter u with the linear interpolation will give us the effect we desire. We can use this approach to change any curve valid over the interval $[0, 1]$ and using u as a parameter to be valid over $[t_0, t_1]$ and using t as a parameter.

Piecewise Linear Interpolation

Pure linear interpolation works fine if we have only two values, but in most cases, we will have many more than two. How do we interpolate among multiple points? The simplest method is to use piecewise curves; that is, we linearly interpolate from the first point to the second, then from the second point to the third, and so on, until we get to the end. For each pair of points P_i and P_{i+1}, we use equation 10.1 to adjust the time range $[t_i, t_{i+1}]$ to $[0, 1]$ so we can interpolate properly.

For a given time value t, we need to find the stored time values t_i and t_{i+1} such that $t_i \le t \le t_{i+1}$. From there we look up their corresponding P_i and P_{i+1} values and interpolate. If we start with $n + 1$ points, we will end up with a series of n segments labeled $Q_0, Q_1, \ldots, Q_{n-1}$. Each Q_i is defined by points P_i and P_{i+1} where

$$Q_i(u) = (1 - u)P_i + uP_{i+1}$$

and $Q_i(1) = Q_{i+1}(0)$. This last condition guarantees C^0 continuity. This is expressed as code as follows:

```
IvVector3 EvaluatePiecewiseLinear( float t, unsigned int count,
                                   const IvVector3* positions,
                                   const float* times)
{
    // handle boundary conditions
    if ( t <= times[0] )
        return positions[0];
    else if ( t >= times[count-1] )
        return positions[count-1];

    // find segment and parameter
    unsigned int i;
    for ( i = 0; i < count-1; ++i )
    {
        if ( t < times[i+1] )
            break;
    }
    float t0 = times[i];
    float t1 = times[i+1];
    float u = (t - t0)/(t1 - t0);

    //evaluate
    return (1-u)*positions[i] + u*positions[i+1];
}
```

In the pseudocode we found the subcurve by using a straight linear search. For large sets of points, using a binary search will be more efficient since we'll be storing the values in sorted order. We can also use temporal coherence: since our time values won't be varying wildly and most likely will be increasing in value, we can first check whether we lie in the interval $[t_i, t_{i+1}]$ from the last frame and then check subsequent intervals.

This works reasonably well and is quite fast, but as Figure 10.1 demonstrates, will lead to sharp changes in direction. If we treat the piecewise

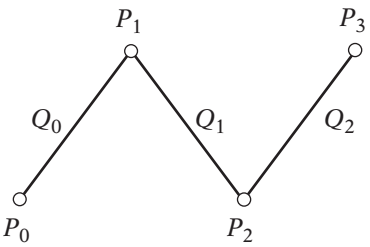


FIGURE 10.1 Piecewise linear interpolation.

interpolation of $n + 1$ points as a single function $f(t)$ over $[t_0, t_n]$, we find that the derivative $f'(t)$ is discontinuous at the sample points, so $f(t)$ is not C^1 continuous. In animation this expresses itself as sudden changes in the speed and direction of motion, which may not be desirable. Despite this, because of its speed, piecewise linear interpolation is a reasonable choice if the slopes of the piecewise line segments are relatively close. If not, or if smoother motion is desired, other methods using higher-order polynomials are necessary.

10.2.3 HERMITE CURVES

Definition

SOURCE CODE DEMO Hermite

The standard method of improving on piecewise linear equations is to use piecewise cubic curves. If we control the curve properly at each point, then we can smoothly transition from one point to the next, avoiding the obvious discontinuities. In particular, what we want to do is to set up our piecewise curves so that the tangent at the end of one curve matches the tangent at the start of the next curve. This will remove the first order discontinuity at each point — the derivative will be continuous over the entire time interval that we are concerned with.

Why a cubic curve and not a quadratic curve? Take a look at Figure 10.2. We have set two positions P_0 and P_1, and two tangents $\mathbf{P}'_0$ and $\mathbf{P}'_1$. Clearly, a line won't pass through the two points and also have a derivative at each point that matches its corresponding tangent vectors. The same is true for a parabola. The next order curve is cubic, which will satisfy these conditions. Intuitively, this makes sense. A line is constrained by two points, or one point and a vector; a parabola can be defined by three points, or by two points and a tangent; and a cubic curve can be defined by four points, or two points and two tangents.

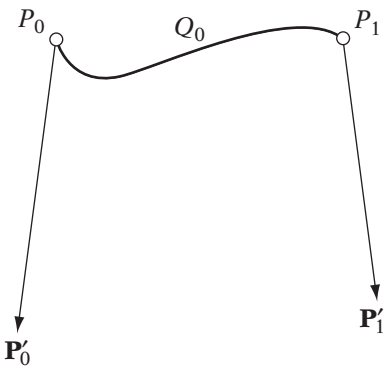


FIGURE 10.2 Hermite curve.

Using our given constraints, or *boundary conditions*, let's derive our cubic equation. A generalized cubic function and corresponding derivative are

$$Q(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + D \tag{10.2}$$

$$\mathbf{Q}'(u) = 3\mathbf{a}u^2 + 2\mathbf{b}u + \mathbf{c} \tag{10.3}$$

We'll solve for our four unknowns **a**, **b**, **c**, and D by using our four boundary conditions. We'll assume that when $u = 0$, $Q(0) = P_0$ and $\mathbf{Q}'(0) = \mathbf{P}'_0$. Similarly, at $u = 1$, $Q(1) = P_1$ and $\mathbf{Q}'(1) = \mathbf{P}'_1$. Substituting these values into equations 10.2 and 10.3, we get

$$Q(0) = D = P_0 \tag{10.4}$$

$$Q(1) = \mathbf{a} + \mathbf{b} + \mathbf{c} + D = P_1 \tag{10.5}$$

$$\mathbf{Q}'(0) = \mathbf{c} = \mathbf{P}'_0 \tag{10.6}$$

$$\mathbf{Q}'(1) = 3\mathbf{a} + 2\mathbf{b} + \mathbf{c} = \mathbf{P}'_1 \tag{10.7}$$

We can see that equations 10.4 and 10.6 already determine that **c** and D are $\mathbf{P}'_0$ and P_0, respectively. Substituting these into equations 10.5 and 10.7 and solving for **a** and **b** gives

$$\mathbf{a} = 2(P_0 - P_1) + \mathbf{P}'_0 + \mathbf{P}'_1$$

$$\mathbf{b} = 3(P_1 - P_0) - 2\mathbf{P}'_0 - \mathbf{P}'_1$$

Substituting our now known values for **a**, **b**, **c**, and D into equation 10.2 gives

$$Q(u) = \left[2(P_0 - P_1) + \mathbf{P}'_0 + \mathbf{P}'_1\right]u^3 + \left[3(P_1 - P_0) - 2\mathbf{P}'_0 - \mathbf{P}'_1\right]u^2 + \mathbf{P}'_0 u + P_0$$

This can be rearranged in terms of the boundary conditions to produce our final equation:

$$Q(u) = (2u^3 - 3u^2 + 1)P_0 + (-2u^3 + 3u^2)P_1 + (u^3 - 2u^2 + u)\mathbf{P}'_0 + (u^3 - u^2)\mathbf{P}'_1$$

This is known as a *Hermite curve*. We can also represent this as the product of a matrix multiplication, just as we did with linear interpolation. In this case, the matrices are

$$\mathbf{U} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

$$\mathbf{M} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} P_0 \\ P_1 \\ \mathbf{P}'_0 \\ \mathbf{P}'_1 \end{bmatrix}$$

We can use either formulation to build piecewise curves just as we did for linear interpolation. As before, we can think of each segment as a separate function, valid over the interval [0, 1]. Then to create a C^1 continuous curve, two adjoining segments Q_i and Q_{i+1} would have to have matching positions such that

$$Q_i(1) = Q_{i+1}(0)$$

and matching tangent vectors such that

$$\mathbf{Q}'_i(1) = \mathbf{Q}'_{i+1}(0)$$

What we end up with is a set of sample positions $\{P_0, \ldots, P_n\}$, tangent vectors $\{\mathbf{P}'_0, \ldots, \mathbf{P}'_n\}$, and times $\{t_0, \ldots, t_n\}$. At a given point adjoining two curve segments Q_i and Q_{i+1},

$$Q_i(1) = Q_{i+1}(0) = P_{i+1}$$
$$\mathbf{Q}'_i(1) = \mathbf{Q}'_{i+1}(0) = \mathbf{P}'_{i+1}$$

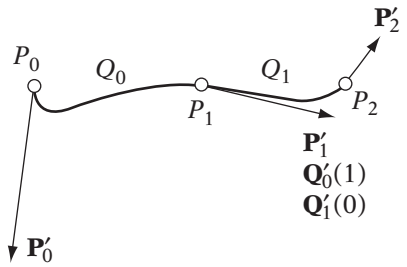


FIGURE 10.3 Piecewise Hermite curve. Tangents at P_1 match direction and magnitude.

Figure 10.3 shows this situation in the piecewise Hermite curve.

The above assumes that our our time values occur at uniform intervals; that is, there is a constant Δt between t_0 and t_1, and t_1 and t_2, etc. However, as mentioned under linear interpolation, the difference between time values t_i to t_{i+1} may vary from segment to segment. The solution is to do the same thing we did for linear interpolation: If we know that a given value t lies between t_i and t_{i+1}, we can use equation 10.1 to normalize our time value to the range $0 \leq u \leq 1$ and use that as our parameter to curve segment Q_i.

This is equivalent to using nonuniform Hermite splines, where the final parameter value is not necessarily equal to 1. These can be derived similarly to the uniform Hermite splines. Assuming a valid range of $[0, t_f]$, their general formula is

$$Q(t) = \left(\frac{2t^3}{t_f^3} - \frac{3t^2}{t_f^2} + 1\right) P_0 + \left(\frac{-2t^3}{t_f^3} + \frac{3t^2}{t_f^2}\right) P_1 + \left(\frac{t^3}{t_f^3} - \frac{2t^2}{t_f^2} + \frac{t}{t_f}\right) \mathbf{P}'_0 + \left(\frac{t^3}{t_f^3} - \frac{t^2}{t_f^2}\right) \mathbf{P}'_1$$

In our case, for each (t_i, t_{i+1}) pair, $t_f = t_{i+1} - t_i$.

Manipulating Tangents

The tangent vectors are used for more than just maintaining first derivative continuity across each sample point. Changing their magnitude also controls the speed at which we move through the point and consequently through the curve. They also affect the shape of the curve. Take a look at Figure 10.4. The longer the vector, the faster we will move and the sharper the curvature. We can create a completely different curve through our sample points, simply by adjusting the tangent vectors.

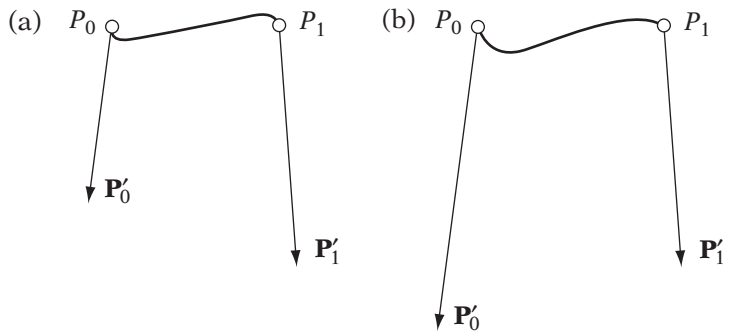


FIGURE 10.4 Hermite curve with (a) small tangent and low curvature and (b) large tangent and higher curvature.

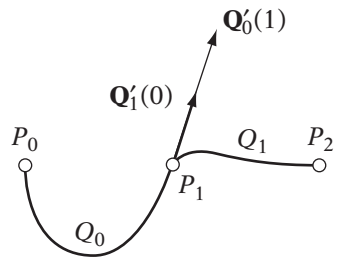


FIGURE 10.5 Piecewise Hermite curve. Tangents at P_1 have same direction but differing magnitudes.

There is, of course, no reason that the tangents $\mathbf{Q}'_i(1)$ and $\mathbf{Q}'_{i+1}(0)$ have to match. One possibility is to match the tangent directions but not the tangent magnitudes—this gives us G^1 continuity. The resulting function has a discontinuity in its derivative but usually still appears smooth. It also has the advantage that it allows us to control how our curve looks across each segment a little better. For example, it might be that we want to have the appearance of a continuous curve but also be able to have more freedom in how each individual segment is shaped. By maintaining the same direction but allowing for different magnitudes, this function provides for the kind of flexibility we need in this instance (Figure 10.5).

Another possibility is that the tangent directions don't match at all. In this case, we'll end up with a kink, or cusp, in the whole curve (Figure 10.6). While not physically realistic, it does allow for sudden changes in direction. The combination of all the possibilities at each sample point—equal tangents, equal tangent directions with nonequal magnitudes, and nonequal tangent directions—gives us a great deal of flexibility in creating our interpolating function across all the sample points. To allow for this level of control, we need

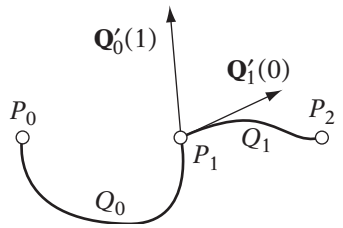


FIGURE 10.6 Piecewise Hermite curve. Tangents at P_1 have differing directions and magnitudes.

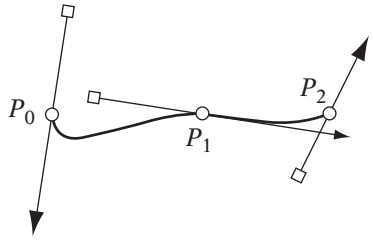


FIGURE 10.7 Possible interface for Hermite curves, showing in–out tangent vectors.

to set two tangents at each internal sample point P_i, which we'll express as $\mathbf{P}'_{i,1}$ (the "incoming" tangent) and $\mathbf{P}'_{i,0}$ (the "outgoing"tangent). Alternatively, we can think of a curve segment as being defined by two points P_i and P_{i+1}, and two tangents $\mathbf{P}'_{i,0}$ and $\mathbf{P}'_{i+1,1}$.

One question remains: How do we generate these tangents? One simple answer is that most existing tools that artists will use, such as Alias's Maya and Discreet's 3D Studio Max, provide ways to set up Hermite curves and their corresponding tangents. When exporting the sample points for subsequent animation, we export the tangents as well. Some tweaking may need to be done to guarantee that the curves generated in internal code match that in the artist program; information on a particular representation is usually available from the manufacturer.

Another common way of generating Hermite data is using in-house tools built for a specific purpose, for example, a tool for managing paths for cameras and other animated objects. In this case, an interface will have to be created to manage construction of the path. One possibility is to click to set the next sample position, and then drag the mouse away from the sample position to set tangent magnitude and direction. A line segment with an arrowhead can be drawn showing the outgoing tangent, and a corresponding line segment with a tail drawn showing the incoming tangent (Figure 10.7).

We will need to modify the tangents so that they can either have different magnitudes or different directions. Many drawing programs control this

by allowing three different tangent types. For example, Jasc's Paint Shop Pro refers to them as symmetric, asymmetric, and cusp. With the symmetric node, clicking and dragging on one of the segment ends rotates both segments and changes their lengths equally, to maintain equal tangents. With an asymmetric node, clicking and dragging will rotate both segments to maintain equal direction but change only the length of the particular tangent clicked on. And with a cusp, clicking and dragging a segment end changes only the length and direction of that tangent. This allows for the full range of possibilities in continuity previously described.

Automatic Generation of Hermite Curves

Suppose we don't need the full control of generating tangents for each sample position. Instead, we just want to automatically generate a smooth curve that passes through all the sample points. To do this, we'll need to have a method of creating reasonable tangents for each sample. One solution is to generate a quadratic function using a given sample point and its two neighbors, and then take the derivative of the function to get a tangent value at the sample point. A similar possibility is to take, for a given point P_i, the weighted average of $(P_{i+1} - P_i)$ and $(P_i - P_{i-1})$. However, for both of these it still will be necessary to set a tangent for the two endpoints, since they have only one neighboring point.

Another method creates tangents that maintain C^2 continuity at the interior sample points. To do this, we'll need to solve a system of linear equations, using our sample points as the known quantities and the tangents as our unknowns. For simplicity's sake, we'll assume we're using uniform curves, and begin by computing the first derivative of the Hermite uniform curve Q:

$$\mathbf{Q}'_i(u) = (6u^2 - 6u)P_i + (-6u^2 + 6u)P_{i+1} + (3u^2 - 4u + 1)\mathbf{P}'_i + (3u^2 - 2u)\mathbf{P}'_{i+1}$$

and from that the second derivative $\mathbf{Q}''$:

$$\mathbf{Q}''_i(u) = (12u - 6)P_i + (-12u + 6)P_{i+1} + (6u - 4)\mathbf{P}'_i + (6u - 2)\mathbf{P}'_{i+1}$$

At a given interior point P_{i+1}, we want the incoming second derivative $\mathbf{P}''_{i+1,1}$ to equal the outgoing second derivative $\mathbf{P}''_{i+1,0}$. We'll assume that each curve segment has a valid parameterization from 0 to 1, so we want

$$\mathbf{Q}''_i(1) = \mathbf{Q}''_{i+1}(0)$$

$$6P_i - 6P_{i+1} + 2\mathbf{P}'_i + 4\mathbf{P}'_{i+1} = -6P_{i+1} + 6P_{i+2} - 4\mathbf{P}'_{i+1} - 2\mathbf{P}'_{i+2}$$

This can be rewritten to place our knowns on one side of the equation and unknowns on the other:

$$2\mathbf{P}'_i + 8\mathbf{P}'_{i+1} + 2\mathbf{P}'_{i+2} = 6[(P_{i+2} - P_{i+1}) + (P_{i+1} - P_i)]$$

This simplifies to

$$\mathbf{P}'_i + 4\mathbf{P}'_{i+1} + \mathbf{P}'_{i+2} = 3(P_{i+2} - P_i)$$

Applying this to all of our sample points $\{P_0, \ldots, P_n\}$ creates $n - 1$ linear equations. This can be written as a matrix product as follows:

$$\begin{bmatrix} 1 & 4 & 1 & & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & 1 & 4 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 4 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \\ \mathbf{P}'_n \end{bmatrix} = \begin{bmatrix} 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \end{bmatrix}$$

This means we have $n-1$ equations with $n+1$ unknowns. To solve this, we will need two more equations. We have already constrained our interior tangents by ensuring C^2 continuity; what remains is to set our two tangents at each extreme point. One possibility is to set them to given values $\mathbf{v}_0$ and $\mathbf{v}_1$, or

$$\mathbf{Q}'_0(0) = \mathbf{P}'_0 = \mathbf{v}_0 \tag{10.8}$$

$$\mathbf{Q}'_{n-1}(1) = \mathbf{P}'_n = \mathbf{v}_1 \tag{10.9}$$

This is known as a clamped end condition, and the resulting curve is a *clamped cubic spline*. Our final system of equations is

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 4 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & 1 & 4 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 4 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \\ \mathbf{P}'_n \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 \\ 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \\ \mathbf{v}_1 \end{bmatrix}$$

Solving this system of equations gives us the appropriate tangent vectors. This is not as bad as it might seem. Because this matrix (known as a *tridiagonal matrix*) is sparse and extremely structured, the system is very easy and efficient to solve using a modified version of Gaussian elimination known as the *Thomas algorithm*.

If we express our tridiagonal matrix generally as

$$\begin{bmatrix} b_0 & c_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & b_1 & c_1 & 0 & \cdots & 0 & 0 \\ 0 & a_2 & b_2 & c_2 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & a_{n-2} & b_{n-2} & c_{n-2} & 0 \\ 0 & 0 & \cdots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & 0 & \cdots & 0 & 0 & a_n & b_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_3 \\ \vdots \\ d_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix}$$

Then we can forward substitute to create array $\mathbf{A}'$ as follows:

$$\begin{aligned} a'_i &= 0 \\ b'_i &= 1 \\ c'_i &= \begin{cases} \frac{c_0}{b_0} & ; \quad i = 0 \\ \frac{c_i}{b_i - c'_{i-1} a_i} & ; \quad 1 \le i \le n-1 \end{cases} \\ d'_i &= \begin{cases} \frac{d_0}{b_0} & ; \quad i = 0 \\ \frac{d_i - d'_{i-1} a_i}{b_i - c'_{i-1} a_i} & ; \quad 1 \le i \le n \end{cases} \end{aligned}$$

Here $\mathbf{A}'$ et al. represent a modification of their respective counterparts, not a derivative.

We can then solve for $\mathbf{x}$ by using back substitution:

$$\begin{aligned} x_n &= d'_n \\ x_i &= d'_i - c'_i x_{i+1} \quad ; \quad 0 \le i \le n-1 \end{aligned}$$

This is significantly faster than blindly applying Gaussian elimination. In addition to the speed-up, we can also use less space than Gaussian elimination by storing our matrix as three $n+1$–length arrays: a, b, and c. So the fact that our matrix is tridiagonal leads to a great deal of savings.

Natural End Conditions

SOURCE CODE DEMO AutoHermite

In the preceding examples, we generated splines assuming that the beginning and end tangents were clamped to values set by the programmer or the user. This may not be convenient; we may want to avoid specifying tangents at all. An alternative approach is to set conditions on the end tangents, just as we did with the internal tangents, to reduce the amount of input needed.

One such possibility is to assume that the second derivative is 0 at the two extremes; that is, $\mathbf{Q}_0''(0) = \mathbf{Q}_{n-1}''(1) = 0$. This is known as a *relaxed* or *natural* end condition, and the spline created is known as a *natural spline*. As the name indicates, this produces a very smooth and natural-looking curve at the endpoints, and in most cases, this is the end condition we would want to use.

With a natural spline, we don't need to specify tangent information at all—we can compute the two unconstrained tangents from the clamped spline using the second derivative condition.

At point P_0, we know that

$$\begin{aligned} 0 &= \mathbf{Q}_0''(0) \\ &= -6P_0 + 6P_1 - 4\mathbf{P}_0' - 2\mathbf{P}_1' \end{aligned}$$

As before, we can rewrite this so that the unknowns are on the left side and the knowns on the right:

$$4\mathbf{P}_0' + 2\mathbf{P}_1' = 6P_1 - 6P_0$$

or

$$2\mathbf{P}_0' + \mathbf{P}_1' = 3(P_1 - P_0) \tag{10.10}$$

Similarly, at point P_n, we know that

$$\begin{aligned} 0 &= \mathbf{Q}_{n-1}''(1) \\ &= 6P_{n-1} - 6P_n + 2\mathbf{P}_{n-1}' + 4\mathbf{P}_n' \end{aligned}$$

This can be rewritten as

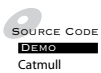
$$\mathbf{P}_{n-1}' + 2\mathbf{P}_n' = 3(P_n - P_{n-1}) \tag{10.11}$$

We can substitute equations 9.12 and 9.13 for our first and last equations in the clamped case, to get the following matrix product:

$$\begin{bmatrix} 2 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 4 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 4 & 1 & \cdots & 0 & 0 \\ & & & \vdots & & & \\ 0 & 0 & \cdots & 1 & 4 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & 4 & 1 \\ 0 & 0 & \cdots & 0 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{P}'_0 \\ \mathbf{P}'_1 \\ \vdots \\ \mathbf{P}'_{n-1} \\ \mathbf{P}'_n \end{bmatrix} = \begin{bmatrix} 3(P_1 - P_0) \\ 3(P_2 - P_0) \\ 3(P_3 - P_1) \\ \vdots \\ 3(P_{n-1} - P_{n-3}) \\ 3(P_n - P_{n-2}) \\ 3(P_n - P_{n-1}) \end{bmatrix}$$

Once again, by solving this system of linear equations, we can find the values for our tangents.

10.2.4 Catmull-Rom Splines



An alternative for automatic generation of a parametric curve is the *Catmull-Rom spline*. This takes a similar approach to some of the initial methods we described for Hermite curves (tangent of parabola, weighted average), where tangents are generated based on the positions of the sample points. The standard Catmull-Rom splines create the tangent for a given sample point by taking the neighboring sample points, subtracting to create a vector, and halving the length. So, for sample P_i, the tangent $\mathbf{P}'_i$ is

$$\mathbf{P}'_i = \frac{1}{2}(P_{i+1} - P_{i-1})$$

If we substitute this into our matrix definition of a Hermite curve between P_i and P_{i+1}, this gives us

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_i \\ P_{i+1} \\ \frac{1}{2}(P_{i+1} - P_{i-1}) \\ \frac{1}{2}(P_{i+2} - P_i) \end{bmatrix}$$

We can rewrite this in terms of P_{i-1}, P_i, P_{i+1}, and P_{i+2} to get

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

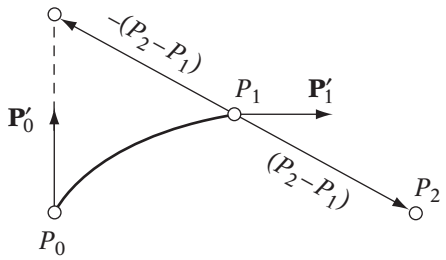


FIGURE 10.8 Automatic generation of tangent vector at P_0, based on positions of P_1 and P_2.

This provides a definition for curve segments Q_1 to Q_{n-2}, so it can be used to generate a C^1 curve from P_1 to P_{n-1}. However, since there is no P_{-1} or P_{n+1}, we once again have the problem that curves Q_0 and Q_{n-1} are not valid due to undefined tangents at the endpoints. And as before, these either can be provided by the artist or programmer, or automatically generated. Parent [88] presents one technique. For P_0, we can take the next two points, P_1 and P_2, and use them to generate a new phantom point, $P_1 + (P_1 - P_2)$. If we subtract P_0 from the phantom point and halve the length, this gives a reasonable tangent for the start of the curve (Figure 10.8). The tangent at P_n can be generated similarly.

Since our knowns for the outer curve segments are two points and a tangent, another possibility is to use a quadratic equation to generate these segments. We can derive this in a similar manner as the Hermite spline equation. The general quadratic equation will have the form

$$Q(u) = \mathbf{a}u^2 + \mathbf{b}u + C \tag{10.12}$$

For the case of Q_0, we know that

$$\begin{aligned} Q_0(0) &= C = P_0 \\ Q_0(1) &= \mathbf{a} + \mathbf{b} + C = P_1 \\ \mathbf{Q}'_0(1) &= 2\mathbf{a} + \mathbf{b} = \mathbf{P}'_1 \\ &= \frac{1}{2}(P_2 - P_0) \end{aligned}$$

Solving for **a**, **b**, and C and substituting into equation 10.12, we get

$$Q_0(u) = \left(\frac{1}{2}P_0 - P_1 + \frac{1}{2}P_2\right)u^2 + \left(-\frac{3}{2}P_0 + 2P_1 - \frac{1}{2}P_2\right)u + P_0$$

Rewriting in terms of P_0, P_1, and P_2 gives

$$Q_0(u) = \left(\frac{1}{2}u^2 - \frac{3}{2}u + 1\right) P_0 + \left(-u^2 + 2u\right) P_1 + \left(\frac{1}{2}u^2 - \frac{1}{2}u\right) P_2$$

As before, we can write this in matrix form:

$$Q_0(u) = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} 1 & -2 & 1 \\ -3 & 4 & -1 \\ 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \end{bmatrix}$$

A similar process can be used to derive Q_{n-1}:

$$Q_{n-1}(u) = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} 1 & -2 & 1 \\ -1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix} \begin{bmatrix} P_{n-2} \\ P_{n-1} \\ P_n \end{bmatrix}$$

10.2.5 Kochanek-Bartels Splines

An extension of Catmull-Rom splines are Kochanek-Bartels splines [66]. Like Catmull-Rom splines, the tangents are generated based on the positions of the sample points. However, rather than generating a single tangent at each point, Kochanek-Bartels splines separate the incoming and outgoing tangents. In addition, rather than using a fixed function based on the preceding and following points, the tangents are computed from a weighted sum of two vectors: the difference between the following and current point $P_{i+1} - P_i$, and the difference between the current point and the preceding point $P_i - P_{i-1}$.

The weights in this case are based on three parameters: tension (represented as τ), continuity (represented as γ), and bias (represented as β). Because of this, they are also often called TCB splines.

The formulae for the tangents at a sample P_i on a Kochanek-Bartels spline are as follows:

$$\mathbf{P}'_{i,0} = \frac{(1-\tau)(1-\gamma)(1-\beta)}{2}(P_{i+1} - P_i) + \frac{(1-\tau)(1+\gamma)(1+\beta)}{2}(P_i - P_{i-1})$$

$$\mathbf{P}'_{i,1} = \frac{(1-\tau)(1+\gamma)(1-\beta)}{2}(P_{i+1} - P_i) + \frac{(1-\tau)(1-\gamma)(1+\beta)}{2}(P_i - P_{i-1})$$

Note that each of these parameters has a valid range of $[-1, 1]$. Also note that if all are set to 0, then we end up with the formula for a Catmull-Rom spline.

Each parameter has a different effect on the shape of the curve. For example, as the tension at a given control point varies from -1 to 1, the curve passing

through the point will change from a very rounded curve to a very tight curve. One can think of it as increasing the influence of the control point on the curve (Figure 10.9(a)).

Continuity does what one might expect—it varies the continuity at the control point. A continuity setting of 0 means that the curve will have C^1 continuity at that point. As the setting approaches -1 or 1, the curve will end up with a corner at that point; the sign of the continuity controls the direction of the corner (Figure 10.9(b)).

Bias varies the effect of P_{i+1} and P_{i-1} on the tangents. A bias near -1 means that $P_{i+1} - P_i$ will have the most effect on the tangents; this is called *undershooting*. If the bias is near 1, then $P_i - P_{i-1}$ will have the most effect; this is called *overshooting* (Figure 10.9(c)).

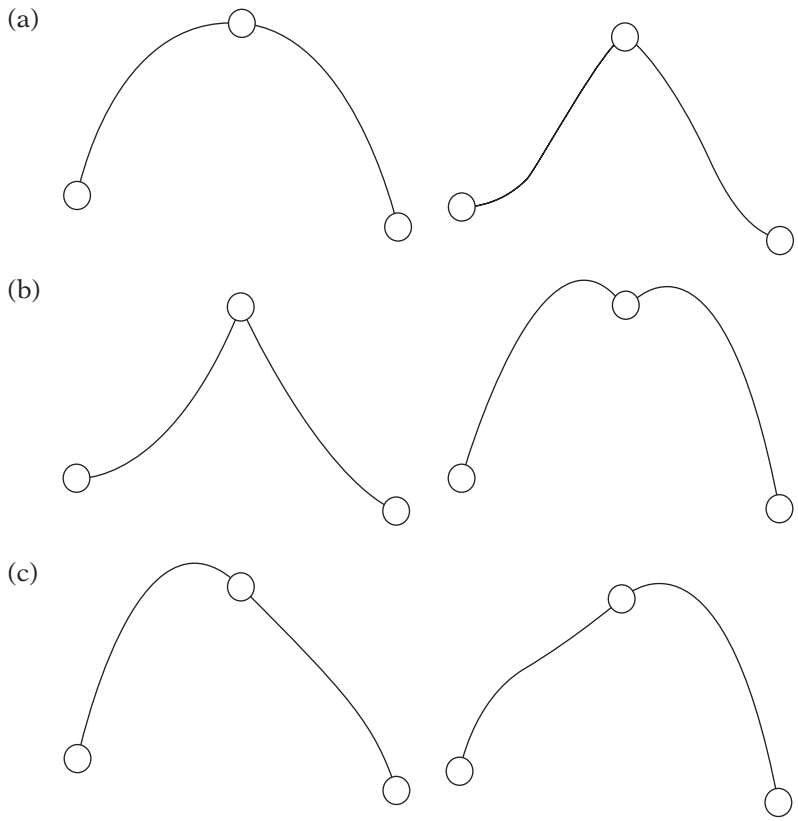


FIGURE 10.9 Kochanek-Bartels curves. (a) Effect of low versus high tension at central point, (b) effect of low versus high continuity at central point, (c) effect of low versus high bias at central point.

Note that these splines have the same problem as Catmull-Rom splines with undefined tangents at the endpoints as there is only one neighboring point. As before, this can be handled by the user setting these tangents by hand or building quadratic curves for the first and last segments. The process for generating these is similar to what we did for Catmull-Rom splines.

Kochanek-Bartels splines are useful because they provide more control over the resulting curve than straight Catmull-Rom splines, and are often used in three-dimensional (3D) packages as an interface to Hermite splines. Because of this, it is useful to be aware of them for use in internal tools and for handling when exporting from commercial software.

10.2.6 Bézier Curves

Definition

SOURCE CODE DEMO Bézier

The previous techniques for generating curves from a set of points meet the functional requirements of controlling curvature and maintaining continuity. However, other than Hermite curves where the tangents are user-specified, they are not so good at providing a means of controlling the shape that is produced. It is not always clear how adjusting the position of a point will change the curve produced, and if we're using a particular type of curve and want to pass through a set of fixed points, there is usually only one possibility.

Bézier curves were created to meet this need. They were devised by Pierre Bézier for modeling car bodies for Renault and further refined by Forrest, Gordon, and Riesenfeld. A cubic Bézier curve uses four *control points*: two endpoints P_0 and P_3 that the curve interpolates, and two points P_1 and P_2 that the curve approximates. Their positions act, as their name suggests, to control the curve. The convex hull, or control polygon, formed by the control points bounds the curve (Figure 10.10). Another way to think of it is that the

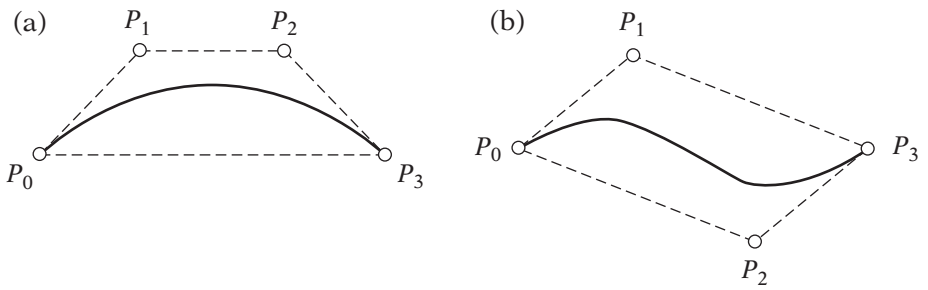


Figure 10.10 Examples of cubic Bézier curve showing convex hull.

curve mimics the shape of the control polygon. Note that the four points in this case do not have to be coplanar, which means that the curve generated will not necessarily lie on a plane either.

The tangent vector at point P_0 points in the same direction as the vector $P_1 - P_0$. Similarly, the tangent at P_3 has the same direction as $P_3 - P_2$. As we will see, there is a definite relationship between these vectors and the tangent vectors used in Hermite curves. For now, we can think of the polygon edge between the interpolated endpoint and neighboring control point as giving us an intuitive sense of what the tangent is like at that point.

So far we've only shown cubic Bézier curves, but there is no reason why we couldn't use only three control points to produce a quadratic Bézier curve (Figure 10.11) or more control points to produce higher-order curves. A general Bézier curve is defined by the function

$$Q(u) = \sum_{i=0}^{n} P_i J_{n,i}(u)$$

where the set of P_i are the control points, and

$$J_{n,i}(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

where

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

The polynomials generated by $J_{n,i}$ are also known as the Bernstein polynomials, or *Bernstein basis*.

In most cases, however, we will use only cubic Bézier curves. Higher-order curves are more expensive and can lead to odd oscillations in the shape of the curve. Quadratic curves are useful when processing power is limited (e.g., the

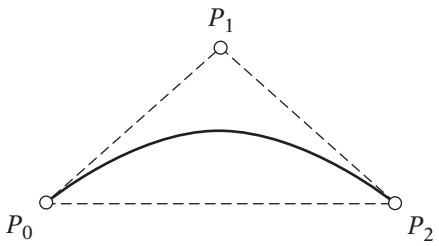


FIGURE 10.11 Example of quadratic Bézier curve showing convex hull.

game Quake 3 used them) but don't have quite the flexibility of cubic curves. For example, they don't allow for the familiar S shape in Figure 10.10(b). To generate something similar with quadratic curves requires two piecewise curves, and hence more data.

The standard representation of an order n Bézier curve is to use an ordered list of points $P_0, \ldots, P_n$ as the control points. Using this representation, we can expand the general definition to get the formula for the cubic Bézier curve:

$$Q(u) = (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u)P_2 + u^3 P_3 \tag{10.13}$$

The matrix form is

$$Q(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

We can think of the curve as a set of affine combinations of the four points, where the weights are defined by the four basis functions $J_{3,i}$. We can see these basis functions graphed in Figure 10.12. At a given parameter value u, we grab the four basis values and use them to compute the affine combination.

As hinted at, there is a relationship between cubic Bézier curves and Hermite curves. If we set our Hermite tangents to $3(P_1 - P_0)$ and $3(P_3 - P_2)$, substitute those values into our cubic Hermite equation, and simplify, we end up with the cubic Bézier equation.

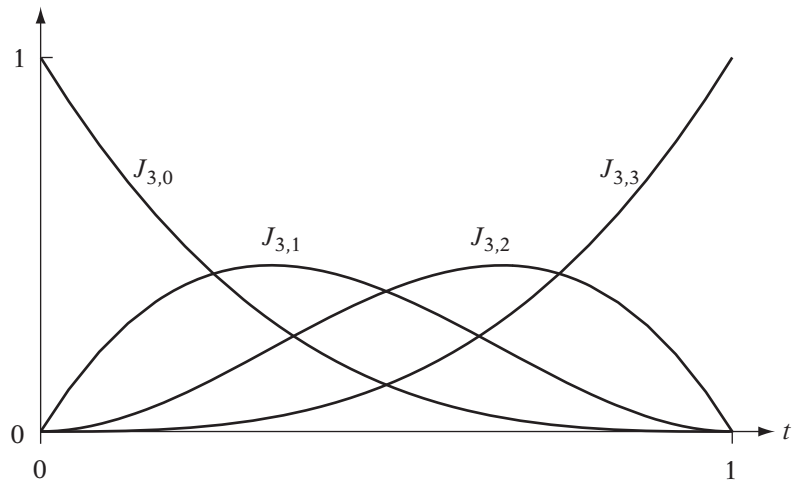


FIGURE 10.12 Cubic Bézier curve basis functions.

Piecewise Bézier Curves

As with linear interpolation and Hermite curves, we can interpolate a curve through more than two points by creating curve segments between each neighboring pair of interpolation points. Many of the same principles apply with Bézier curves as did with Hermite curves. In order to maintain matching direction for our tangents, giving us G^1 continuity, each interpolating point and its neighboring control points need to be collinear. To obtain equal tangents, and therefore C^1 continuity, the control points need to be collinear with and equidistant to the shared interpolating point. Drawing a line segment through the three points gives a three-lobed barbell shape, seen in Figure 10.13.

The barbell makes another very good interface for managing our curves. If we set up our interpolating point as a pivot, then we can grab one neighboring control point and rotate it around to change the direction of the tangent. The other neighboring control point will rotate correspondingly to maintain collinearity and equal distance, and thereby C^1 continuity. If we drag the control point away from our interpolating point, that will increase the length of our tangent. We can leave the other control point at the original distance, if we like, to create different arrival/departure speeds while still maintaining G^1 continuity. Or, we can match its distance from the sample as well, to maintain C^1 continuity. And of course, we can move each neighboring control point independently to create a cusp at that interpolating point.

This seems very similar to our Hermite interface, so the question may be, why use Bézier curves? The main advantage of the Bézier interface over the Hermite interface is that, as mentioned, the control points act to bound the curve, and so give a much better idea of how the shape of the curve will change as we move the control points around. Because of this, many drawing packages use Bézier curves instead of Hermite curves.

While in most cases we will want to make use of user-created data with Bézier curves, it is sometimes convenient to automatically generate them. One possibility is to use the modification of the matrix technique we used with Hermite curves. Alternatively, Parent [88] provides a method for

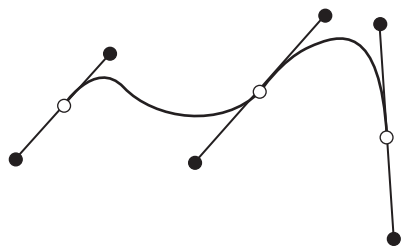


FIGURE 10.13 Example interface for Bézier curves.

automatically generating Bézier control points from a set of sample positions, as shown in Figure 10.14. Given four points P_{i-1}, P_i, P_{i+1}, and P_{i+2}, we want to compute the two control points between P_i and P_{i+1}. We compute the tangent vector at P_i by computing the difference between P_{i+1} and P_{i-1}. From that we can compute the first control point as $P_i + 1/3(P_{i+1} - P_{i-1})$. The same can be done to create the second control point as $P_{i+1} - 1/3(P_{i+2} - P_i)$. This is very similar to how we created the Catmull-Rom spline, but with tangents twice as large in magnitude.

10.2.7 Other Curve Types

Source Code DEMO B-Spline

The first set of curves we looked at were interpolating curves, which pass through all the given points. With Bézier curves, the resulting curve interpolates two of the control points, while approximating the others. *B-splines* are a generalization of this — depending on the form of the B-spline, all or none of the points can be interpolated. Because of this, in a B-spline all of the control points can be used as approximating points (Figure 10.15). In fact, B-splines are so flexible they can be used to represent all of the curves we have described so far. However, with flexibility comes a great deal of complexity. Because of

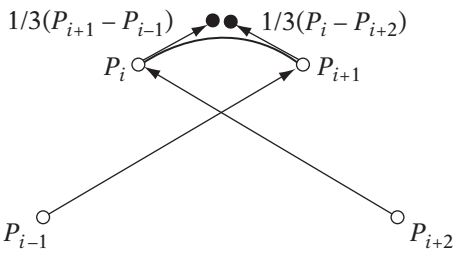


Figure 10.14 Automatic construction of approximating control points with Bézier curve.

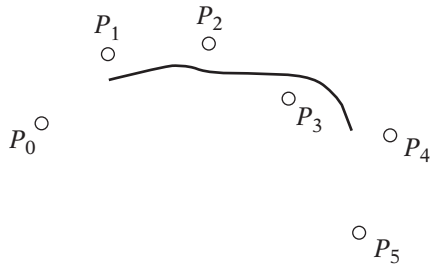


Figure 10.15 B-spline approximating curve.

this, B-splines are not yet in common usage in games, either for animation or surface construction.

B-splines are computed similarly to Bézier curves. We set up a basis function for each control point in our curve, and then for each parameter value u, we multiply the appropriate basis function by its point and add the results. In general, this can be represented by

$$Q(u) = \sum_{i=0}^{n} P_i B_i(u)$$

where each P_i is a point and B_i is a basis function for that point. The basis functions in this case are far more general than those described for Bézier curves, which gives B-splines their flexibility and their power.

Like our previous piecewise curves, B-splines are broken into smaller segments. The difference is that the number of segments is not necessarily dependent on the number of points, and the intermediary point between each segment is not necessarily one of our control points. These intermediary points are called *knots*. If the knots are spaced equally in time, the curve is known as a *uniform B-spline*, otherwise it is a *nonuniform B-spline*.

B-splines are not often used for animation; they are more commonly used when building surface representations. A full description of the power and complexity of B-splines is out of the purview of this text, so for those who are interested, more information on B-splines and other curves can be found in Bartels et al. [6], Foley et al. [38], and Rogers [97].

Another issue is that the curves we have discussed so far have the property that any affine transformation on the set of points (or tangents, in the case of Hermite curves) generating the curve will transform the curve accordingly. So for example, if we want to transform a Bézier curve from the local frame to the view frame, all we need to do is transform the control points and then generate the curve in the view frame.

However, this will not work for a perspective transformation, due to the need for a reciprocal division at each point on the curve. The answer is to apply a process similar to the one we used when transforming points, by adding an additional parameterized function $w(u)$ that we divide by when generating the points along the curve. This is known as a *rational curve*.

There are a number of uses for rational curves. The first has already been stated: We can use it as a more efficient method for projecting curves. But it also allows us to set weights w_i for the control points so that we can direct the curve to pass closer to one point or another. Another use of rational curves is to create conic section curves, such as circles and ellipses. Nonrational curves, since they are polynomials, can only approximate conic sections.

The most commonly used of the rational curves are nonuniform rational B-splines, or NURBS. Since they can produce conic as well as general curves

and surfaces, they are extremely useful in computer-aided design (CAD) systems and modeling for computer animation. Like B-splines, rational curves and particularly NURBS are not yet used much in games because of their relative performance cost and because of concern by artists about lack of control.

10.3 Interpolation of Orientation

So far in our exploration of animation we've considered only interpolation of position. For a coordinate frame, this means only translating the frame in space, without considering rotation. This is fine for moving an object along a path, assuming we wanted it to remain oriented in the same manner as its base frame, however, generally we don't. One possibility is to align the forward vector of the object to the tangent vector of the curve, and use either the second derivative vector or an up vector to build a frame. This will work in general for airplanes and missiles, which tend to orient along their direction of travel. But suppose we want to interpolate a camera so that it travels sideways along a section of curve, or we're trying to model a helicopter, which can face in one direction while moving in another.

Another reason we want to interpolate orientation is for the purpose of animating a character. Usually characters are broken into a scene-graph–like data structure, called the *skeleton*, where each level, or *bone*, is stored at a constant translation from its parent, and only relative rotation is changed to move a particular node (Figure 10.16). So to move a forearm, for example, we rotate it relative to an upper arm (Figure 10.17). Accordingly, we can generate a set of key frames for an animated character by storing a set of poses generated by setting rotations at each bone. To animate the character, we interpolate from one key frame rotation to another.

As we shall see, when interpolating orientation we can't quite use the same techniques as we did with position. Rotational space doesn't behave in the same way as positional space; we'll be more concerned with interpolating along the surface of a sphere instead of along a line. As part of this, we'll revisit the representations we covered in Chapter 5 discussing the pros and cons of each representation for handling the task of interpolation.

10.3.1 General Discussion

Our interpolation problem for position was to find a space curve — a function given a time parameter that returns a position — that passes through our sample points and maintains our desired curvature at each sample point. The

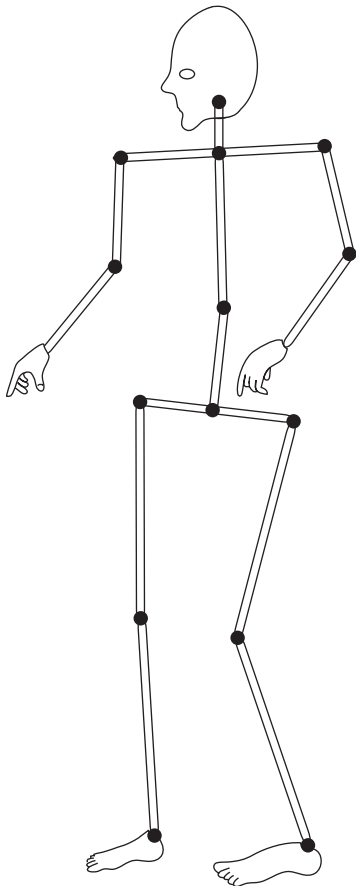


FIGURE 10.16 Example of skeleton showing relationship between bones.

same is true of interpolating orientation, except that our curve doesn't pass through a series of positions, but a series of orientations.

We can think of this as wanting to interpolate from one coordinate frame to another. If we were simply interpolating two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, we could find the rotation between them via the axis–angle representation $(\theta, \hat{\mathbf{r}})$, and then interpolate by rotating $\mathbf{v}_1$ as

$$\mathbf{v}(t) = R(t\theta, \hat{\mathbf{r}})\,\mathbf{v}_1$$

In other words, we linearly interpolate the angle from 0 to θ and continually apply the newly generated rotation to $\mathbf{v}_1$ to get our interpolated orientations. But for a coordinate frame, we need to interpolate three vectors

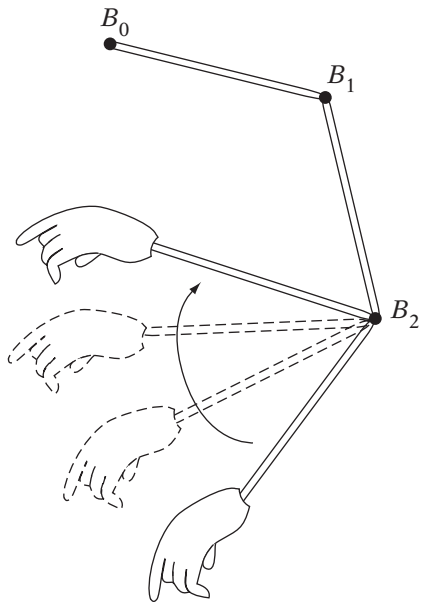


FIGURE 10.17 Relative bone poses for bending arm.

simultaneously. We could use the same process for all three basis vectors, but it's not guaranteed that they will remain orthogonal. What we would need to do is find the overall rotation in axis–angle form from one coordinate frame to another, and then apply the process described. This is not a simple thing to do, and as it turns out, there are better ways.

However, for fixed angles and axis–angle formats, we can use this to interpolate simple cases of rotation around a single axis. For instance, if we're interpolating from (90, 0, 0) to (180, 0, 0), we can linearly interpolate the first angle from 90 degrees to 180 degrees. Or, with an axis–angle format, if the rotation is from the reference orientation to another orientation, again we only need to interpolate the angle. Using this method also allows for interpolations over angles greater than 360 degrees. Suppose we want to rotate twice around the z-axis and represent this as only two values. We could interpolate between the two x-y-z fixed angles (0, 0, 0) and (0, 0, 4π). As we interpolate from 0 to 1, our object will rotate twice. More sample orientations are needed to do this with matrices and quaternions.

SOURCE CODE DEMO Euler

But extending this to more complex cases does not work. Suppose we take as our starting orientation (0, 90, 0) and our ending orientation (90, 45, 90). If we linearly interpolate the angles to find a value halfway between them, we get (45, 67.5, 45). But this is wrong. One possible value that is correct is

(90, 22.5, 90). The consequence of interpolating linearly from one sequence of Euler angles to another is that the object tends to sidle along, rotating around mostly one axis and then switching to rotations around mostly another axis, instead of rotating around a single axis, directly from one orientation to another.

We can mitigate this problem by defining Hermite or higher-order splines to better control the interpolation, and some 3D modeling packages provide output to do just that. However, you may not want to dedicate the space for the intermediary key frames or the processing power to perform the spline interpolation, and it's still an approximation. For more complex cases, the only two formats that are practical are matrices and quaternions, and as we'll see, this is where quaternions truly shine.

There are generally two approaches used when interpolating matrices and quaternions in games: linear interpolation and spherical linear interpolation. Both methods are usually applied piecewise between each orientation sample pair, and even though this will generate discontinuities at the sample points, the artifacts are rarely noticeable. While we will mention some ways of computing cubic curves, they generally are just too expensive for the small gain in visual quality.

10.3.2 Linear Interpolation

By using the scalar multiplication and addition operations, we can linearly interpolate rotation matrices and quaternions just as we did vectors. Let's look at a matrix example first. Consider two orientations: one represented as the identity matrix and the other by a rotation of 90 degrees around the z-axis. Using linear interpolation to find the orientation halfway between the start and end orientations, we get

$$\frac{1}{2}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} + \frac{1}{2}\begin{bmatrix}0 & 1 & 0\\ -1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}\frac{1}{2} & \frac{1}{2} & 0\\ -\frac{1}{2} & \frac{1}{2} & 0\\ 0 & 0 & 1\end{bmatrix}$$

The result is not a well-formed rotation matrix. The basis vectors are indeed perpendicular, but they are not unit length. In order to restore this, we need to perform Gram-Schmidt orthogonalization, which is a rather expensive operation to perform every time we want to perform an interpolation.

With quaternions we run into some problems similar to those encountered with matrices. Suppose we perform the same interpolation, from the identity

quaternion to a rotation of 90 degrees around z. This second quaternion is $(\sqrt{2}/2, 0, 0, \sqrt{2}/2)$. The resulting interpolated quaternion when $t = 1/2$ is

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(1, 0, 0, 0) + \frac{1}{2}\left(\frac{\sqrt{2}}{2}, 0, 0, \frac{\sqrt{2}}{2}\right) \\ &= \left(\frac{2+\sqrt{2}}{4}, 0, 0, \frac{\sqrt{2}}{4}\right)\end{aligned}$$

The length of $\mathbf{r}$ is 0.9239—clearly, not 1. Just as with matrices, we had to reorthogonalize after performing linear interpolation; with quaternions we will have to renormalize. Fortunately, this is a cheaper operation than orthogonalization, so quaternions have the advantage here.

In both cases, this happens because linear interpolation has the effect of cutting across the arc of rotation. If we compare a vector in one orientation with its equivalent in the other, we can get some sense of this. In the ideal case, as we rotate from one vector to another, the tips of the interpolated vectors trace an arc across the surface of a sphere (Figure 10.18). But as we can see in Figure 10.19, the linear interpolation is following a line segment between the two tips of the vectors, which causes the interpolated vectors to shrink to a length of 1/2 at the halfway point, and then back up to 1.

Another problem with linear interpolation is that it doesn't move at a constant rate of rotation. Let's divide our interpolation at the t values 0, 1/4, 1/2, 3/4, and 1. In the ideal case, we'll travel one-quarter of the arc length to get from orientation to orientation.

However, when we use linear interpolation, the t value doesn't interpolate along the arc, but along the chord that passes between the start and end orientations. When we divide the chord into four equal parts, the corresponding arcs on the surface of the sphere are no longer equal in length (Figure 10.20).

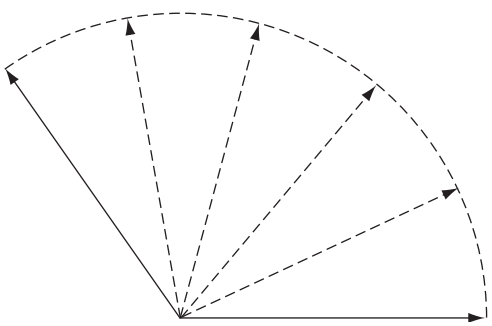


Figure 10.18 Ideal orientation interpolation, showing intermediate vectors tracing path along arc.

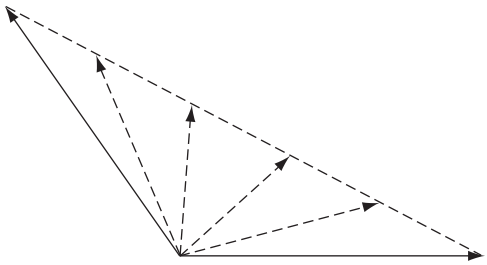


FIGURE 10.19 Linear orientation interpolation, showing intermediate vectors tracing path along line.

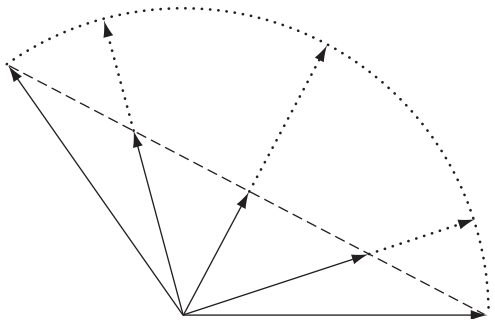


FIGURE 10.20 Effect of linear orientation interpolation on arc length when interpolating over 1/4 intervals.

Those closest to the center of interpolation are longer. The effect is that instead of moving at a constant rate of rotation throughout the interpolation, we will move at a slower rate at the endpoints and faster in the middle. This is particularly noticeable for large angles, as the figure shows. What we really want is a constant change in rotation angle as we apply a constant change in t.

One way to solve both of these issues is to insert one or two additional sample orientations and use quadratic or cubic interpolation. However, these are still only approximations to the spherical curve, and they involve storing additional orientation key frames.

Even if you are willing to deal with nonconstant rotation speed, and eat the cost of orthogonalization, linear interpolation does create other problems. Suppose we use linear interpolation to find the orientation midway between these two matrices:

$$\frac{1}{2}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{10.14}$$

This is clearly not a rotation matrix, and no amount of orthogonalization will help us. The problem is that our two rotations (a rotation of $\pi/2$ around y and a rotation of $-\pi/2$ around y, respectively) produce opposing orientations—they're 180 degrees apart. As we interpolate between the pairs of transformed **i** and **k** basis vectors, we end up passing through the origin.

Quaternions are no less susceptible to this. Suppose we have a rotation of π radians counterclockwise around the y-axis, and a rotation of π radians clockwise around y. Interpolating the equivalent quaternions gives us

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(0, 0, 1, 0) + \frac{1}{2}(0, 0, -1, 0) \\ &= (0, 0, 0, 0)\end{aligned}$$

Again, no amount of normalization will turn this into a unit quaternion. The problem here is that we are trying to interpolate between two quaternions that are negatives of each other. They represent two rotations in the opposite direction that rotate to the *same* orientation. Rotating a vector 180 degrees counterclockwise around y will end up in the same place as rotating the same vector 180 degrees clockwise (or -180 degrees counterclockwise) around y. Even if we considered this an interpolation that runs entirely around the sphere, it is not clear which path to take—there are infinitely many.

This problem with negated quaternions shows up in other ways. Let's look at our first example again, interpolating from the identity quaternion to a rotation of $\pi/2$ around z. Recall that our result with $t = 1/2$ was $(2 + \sqrt{2}/4, 0, 0, \sqrt{2}/4)$. This time we'll negate the second quaternion, giving us a rotation of $-3\pi/2$ around z. We get the result

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(1, 0, 0, 0) + \frac{1}{2}\left(-\frac{\sqrt{2}}{2}, 0, 0, -\frac{\sqrt{2}}{2}\right) \\ &= \left(\frac{2 - \sqrt{2}}{4}, 0, 0, -\frac{\sqrt{2}}{4}\right)\end{aligned}$$

This new result is not the negation of the original result, nor is it the inverse. What is happening is that instead of interpolating along the shortest arc along the sphere, we're interpolating all the way around the other way, via the longest arc. This will happen when the dot product between the two quaternions is negative, so the angle between them is greater than 90 degrees.

This may be the desired result, but usually it's not. What we can do to counteract it is to negate the first quaternion and reinterpolate. In our example, we end up with

$$\begin{aligned}\mathbf{r} &= \frac{1}{2}(-1, 0, 0, 0) + \frac{1}{2}\left(-\frac{\sqrt{2}}{2}, 0, 0, -\frac{\sqrt{2}}{2}\right) \\ &= \left(-\frac{2+\sqrt{2}}{4}, 0, 0, -\frac{\sqrt{2}}{4}\right)\end{aligned}$$

This gives us the negation of our original result, but this isn't a problem as it will rotate to the same orientation.

This also takes care of the case of interpolating from a quaternion to its negative, so for example, interpolating from $(0, 0, 1, 0)$ to $(0, 0, -1, 0)$ is

$$\begin{aligned}\mathbf{r} &= -\frac{1}{2}(0, 0, 1, 0) + \frac{1}{2}(0, 0, -1, 0) \\ &= (0, 0, -1, 0)\end{aligned}$$

Negating the first one ends up interpolating to and from the same quaternion, which is a waste of processing power, but won't give us invalid results. Note that we will have to do this even if we are using spherical linear interpolation, which we will address next. All in all, it is better to avoid such cases by culling them out of our data beforehand.

10.3.3 Spherical Linear Interpolation

To better solve the nonconstant rotation speed and normalization issues, we need an interpolation method known as *spherical linear interpolation* (usually abbreviated as *slerp*[2]). Slerp is similar to linear interpolation except that instead of interpolating along a line, we're interpolating along an arc on the surface of a sphere. Figure 10.21 shows the desired result. When using spherical interpolation at quarter intervals of t, we travel one-quarter of the arc length to get from orientation to orientation. We can also think of slerp as interpolating along the angle, or in this case, dividing the angle between the orientations into quarter intervals.

One interesting aspect of orientations is that operations appropriate for positions move up one step in complexity when applied to orientations. For example, to concatenate positions we add, whereas to concatenate orientations we multiply. Subtraction becomes division, and scalar multiplication becomes exponentiation. Using this knowledge, we can take our linear interpolation function for two rotations P and Q,

$$\text{lerp}\,(P, Q, t) = P + (P - Q)t$$

2. As Shoemake [103] says, because it's fun.

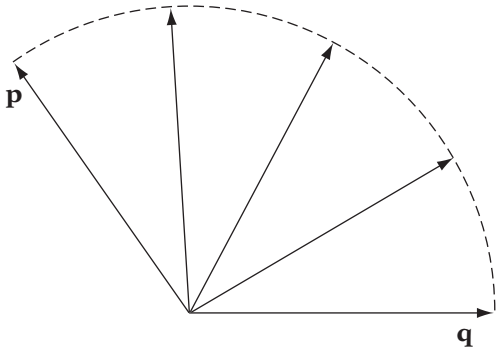


FIGURE 10.21 Effect of spherical linear interpolation when interpolating at quarter intervals. Interpolates equally along arc and angle.

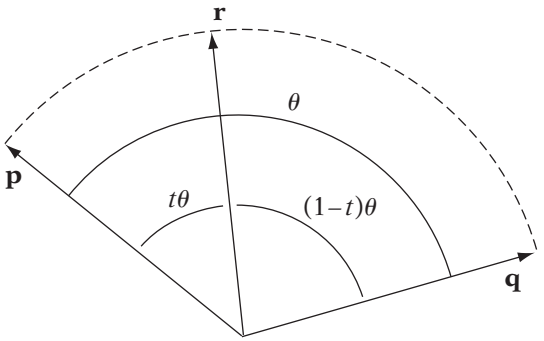


FIGURE 10.22 Construction for quaternion slerp. Angle θ is divided by interpolant t into subangles $t\theta$ and $(1-t)\theta$.

and convert it to the slerp function,

$$\text{slerp}\,(P, Q, t) = P(P^{-1}Q)^t$$

For matrices, the question is how to take a matrix **R** to a power t. We can use a method provided by Eberly [26] as follows. Since we know that **R** is a rotation matrix, we can pull out the axis **v** and angle θ of rotation for the matrix as we've described, multiply θ by t to get a percentage of the rotation, and convert back to a matrix to get $\mathbf{R}^t$. This is an extraordinarily expensive operation. However, if we want to use matrices, it does give us the result we want of interpolating smoothly along arc length from one orientation to another.

For quaternions, we can derive slerp in another way. Figure 10.22 shows the situation. We have two quaternions **p** and **q**, and an interpolated

quaternion $\mathbf{r}$. The angle between $\mathbf{p}$ and $\mathbf{q}$ is θ, calculated as $\theta = \arccos(\mathbf{p} \cdot \mathbf{q})$. Since slerp interpolates the angle, the angle between $\mathbf{p}$ and $\mathbf{r}$ will be a fraction of θ as determined by t, or $t\theta$. Similarly, the angle between $\mathbf{r}$ and $\mathbf{q}$ will be $(1 - t)\theta$.

The general interpolation of $\mathbf{p}$ and $\mathbf{q}$ can be represented as

$$\mathbf{r} = a(t)\,\mathbf{p} + b(t)\,\mathbf{q} \tag{10.15}$$

The goal is to find two interpolating functions $a(t)$ and $b(t)$ so that they meet the criteria for slerp.

We determine these as follows. If we take the dot product of $\mathbf{p}$ with equation 10.15 we get

$$\begin{aligned} \mathbf{p} \cdot \mathbf{r} &= a(t)\,\mathbf{p} \cdot \mathbf{p} + b(t)\,\mathbf{p} \cdot \mathbf{q} \\ \cos(t\theta) &= a(t) + b(t)\cos\theta \end{aligned}$$

Similarly, if we take the dot product of $\mathbf{q}$ with equation 10.15 we get

$$\cos((1 - t)\theta) = a(t)\cos\theta + b(t)$$

We have two equations and two unknowns. Solving for $a(t)$ and $b(t)$ gives us

$$\begin{aligned} a(t) &= \frac{\cos(t\theta) - \cos((1 - t)\theta)\cos\theta}{(1 + \cos^2\theta)} \\ b(t) &= \frac{\cos((1 - t)\theta) - \cos(t\theta)\cos\theta}{(1 + \cos^2\theta)} \end{aligned}$$

Using trigonometric identities, these simplify to

$$\begin{aligned} a(t) &= \frac{\sin((1 - t)\theta)}{\sin\theta} \\ b(t) &= \frac{\sin(t\theta)}{\sin\theta} \end{aligned}$$

Our final slerp equation is

$$\text{slerp}(\mathbf{p}, \mathbf{q}, t) = \frac{\sin((1 - t)\theta)\,\mathbf{p} + \sin(t\theta)\,\mathbf{q}}{\sin\theta} \tag{10.16}$$

As we can see, this still is an expensive operation, consisting of three sines and a floating-point divide, not to mention the precalculation of the arccosine. But at 16 multiplications, 8 additions, 1 divide, and 4 transcendentals,

it is much cheaper than the matrix method. It is clearly preferable to use quaternions versus matrices (or any other form) if you want to interpolate orientation.

One thing to notice is that as θ approaches 0 (i.e., as $\mathbf{p}$ and $\mathbf{q}$ become close to equal) $\sin\theta$ and thus the denominator of the slerp function approach 0. Testing for equality is not enough to catch this case, because of finite floating-point precision. Instead, we should test $\cos\theta$ before proceeding. If it's close to 1 ($> (1 - \epsilon)$, say), then we use linear interpolation or *lerp* instead, since it's reasonably accurate for small angles and avoids the undesirable case of dividing by a very small number. It also has the nice benefit of helping our performance; lerp is much cheaper. In fact, it's generally best only to use slerp in the cases where it is obvious that rotation speed is changing.

Just as we do with linear interpolation, if we want to make sure that our path is taking the shortest route on the sphere and to avoid problems with opposing quaternions, we also need to test $\cos\theta$ to ensure that it is greater than 0 and negate the start quaternion if necessary. While slerp does maintain unit length for quaternions, it's still useful to normalize afterwards to handle any variation due to floating-point error.

Cubic Methods

Just as with lerp, if we do piecewise slerp we will have discontinuities at the sample orientations, which may lead to visible changes in orientation rather than the smooth curve we want. And just as we had available when interpolating points, there are cubic methods for interpolating quaternions. One such method is *squad*, which uses the formula

$$\text{squad}(\mathbf{p}, \mathbf{a}, \mathbf{b}, \mathbf{q}, t) = \text{slerp}(\text{slerp}(\mathbf{p}, \mathbf{q}, t), \text{slerp}(\mathbf{a}, \mathbf{b}, t), 2(1-t)t) \qquad (10.17)$$

This is a modification of a technique of using linear interpolation to do Bézier curves, described by Boehm [13]. It performs a Bézier interpolation from $\mathbf{p}$ to $\mathbf{q}$, using $\mathbf{a}$ and $\mathbf{b}$ as additional control points (or control orientations, to be more precise).

We can use similar techniques for other curve types, such as B-splines and Catmull-Rom curves. However, these methods usually are not used in games. They are more expensive than slerp (which is expensive enough), and most of the time the data being interpolated have been generated by an animation package or exist as samples from motion capture. Both of these tend to smooth the data out and insert additional samples at places where orientation is changing sharply, so smoothing the curve isn't that necessary. For those who are interested, Shoemake [103, 104] covers some of these spline methods in more detail.

10.3.4 Performance Improvements

As we've seen, using slerp for interpolation, even when using quaternions, can take quite a bit of time — something we don't usually have. A typical character can have 30+ bones, all of which are being interpolated once a frame. If we have a team of characters in a room, there can be up to 20 characters being rendered at one time. The less time we spend interpolating, the better.

The simplest speed-up is to use lerp all the time. It's very fast: Ignoring the set-up time (checking angles and adjusting quaternions) and normalization, only 12 basic floating-point operations are necessary on a serial processor, and on a vector processor this drops to 3. We do have the problems with inconsistent rotational speeds, but for animation data where our angles are usually less than 90 degrees, the error is not visually apparent. So in most cases, lerp is a fine solution.

However, if we need to interpolate angles larger than 90 degrees or we are truly concerned with accurate orientations, then we need to try something else. One solution is to improve the speed of slerp. If we assume that we're dealing with a set of stored quaternions for key-framed animation, there are some things we can do here. First of all, we can precompute θ and $1/\sin\theta$ for each quaternion pair and store them with the rest of our animation data. In fact, if we're willing to give up the space, we could prescale $\mathbf{p}$ and $\mathbf{q}$ by $1/\sin\theta$ and store those values instead. This would mean storing up to two copies for each quaternion: one as the starting orientation of an interpolation and one as the ending orientation. Finally, if t is changing at a constant rate, we can use forward differencing to reduce our operations further. Shoemake [104] states that this can be done in 8 multiplies, 6 adds, and 2 table lookups for the two remaining sines.

If memory is plentiful and our frame rate is constant, then this approach can work well. However, neither of these is typically the case. Animation data usually takes up enough of our memory budget without nearly doubling its size, and frame rates can be variable, depending on what is being rendered or simulated. One possibility that doesn't have these restrictions is to approximate the most expensive operations — $1/\sin\theta$, $\sin(t\theta)$, and $\sin((1-t)\theta)$ — by splines. This can provide reasonable accuracy for less cost than the standard evaluation.

An alternate method is proposed by Blow [10]. His idea is that instead of trying to change our interpolation method to fix our variable rotation speeds, we adjust our t values to counteract the variations. So, in the section where an object would normally rotate faster with a constantly increasing t, we slow t down. Similarly, in the section where an object would rotate slower, we speed t up. Blow uses a cubic spline to perform this adjustment:

$$t' = 2kt^3 - 3kt^2 + (1+k)t$$

where

$$k = 0.5069269(1 - 0.7878088\cos\theta)^2$$

and $\cos\theta$ is the dot product between the two quaternions. This technique tends to diverge from the slerp result when $t > 0.5$, so Blow recommends detecting this case and swapping the two quaternions (i.e., interpolate from **q** to **p** instead of from **p** to **q**). In this way our interpolant always lies between 0 and 0.5.

The nice thing about this method is that it requires very few floating-point operations, doesn't involve any transcendental functions or floating-point divides, and fits in nicely with our existing lerp functions. It gives us slerp interpolation quality with close to lerp speed, which can considerably speed up our animation system.

Further possibilities are provided by Busser [15], who approximates $a(t)$ and $b(t)$ in equation 10.15 by polynomial equations, and Thomason [111], who explores a variety of techniques. Whether these would be necessary would depend on your data, although in practice we've found Blow's approach to be sufficient.

10.4 Sampling Curves

Given a parametric curve, it is only natural that we might want to determine a point on it, or *sample* it. We've already stated one reason when motivating interpolation in the introduction: We may have created a curve from a low-resolution set of points, and now want to resample at a higher resolution to match frame rate or simply to provide a better quality animation. Another purpose is to sample the curve at various points, or tesselate it, so that it might be rendered. After all, artists will want to see, and thus more accurately control, the animation paths that they are creating. Finally, we may also want to sample curves for length calculations, as we'll see later.

Sampling piecewise linear splines is straightforward. For rendering we can just draw lines between the sample points. For animation, the function `EvaluatePiecewiseLinear` will do just fine in computing the 'tween points. A similar approach works well when slerping piecewise quaternion curves.

Things get more interesting when we use a cubic curve. For simplicity's sake, we'll only consider one curve segment Q and a parameter u within that segment — determining those are similar to our linear approach. The most direct method is to take the general function for our curve segment $Q(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + D$ and evaluate it at our u values. Assuming that we're generating points in $\mathbb{R}^3$, this will take 11 multiplies and 9 adds per point (we save 3 multiplies by computing u^3 as $u \cdot u^2$).

An alternative that is slightly faster is to use Horner's rule, which expresses the same cubic curve as

$$Q(u) = ((\mathbf{a}u + \mathbf{b})u + \mathbf{c})u + D$$

This will take only 9 multiplies and 9 adds per point. In addition, it can actually improve our floating-point accuracy under certain circumstances.

10.4.1 Forward Differencing

Previously we assumed that there is no pattern to how we evaluate our curve. Suppose we know that we want to sample our curve at even intervals of u, say at a time step of every h. This gives us a list of $n + 1$ parameter values: $0, h, 2h, \ldots, nh$. In such a situation, we can use a technique called *forward differencing*.

For the time being, let's consider computing only the x values for our points. For a given value x_i, located at parameter u, we can compute the next value x_{i+1} at parameter $u + h$. Subtracting x_i from x_{i+1}, we get

$$x_{i+1} - x_i = x(u + h) - x(u)$$

We'll label this difference between x_{i+1} and x_i as $\Delta x_1(u)$. For a cubic curve this equals

$$\begin{aligned}
\Delta x_1(u) &= a(u+h)^3 + b(u+h)^2 + c(u+h) + d - (au^3 + bu^2 + cu + d) \\
&= a(u^3 + 3hu^2 + 3h^2u + h^3) + b(u^2 + 2hu + h^2) + c(u+h) + d \\
&\quad - au^3 - bu^2 - cu - d \\
&= au^3 + 3ahu^2 + 3ah^2u + ah^3 + bu^2 + 2bhu + bh^2 + cu + ch + d \\
&\quad - au^3 - bu^2 - cu - d \\
&= 3ahu^2 + 3ah^2u + ah^3 + 2bhu + bh^2 + ch \\
&= (3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch)
\end{aligned}$$

Pseudocode to compute the set of values might look like the following:

```
u = 0;
x = d;
output(x);
dx1 = ah^3 + bh^2 + ch;
for ( i = 1; i <= n; i++ )
```

```
{
    u += h;
    x += dx1;
    output(x);
    dx1 = (3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch);
}
```

While we have removed the cubic equation, we have introduced evaluation of a quadratic equation $\Delta x_1(u)$. Fortunately, we can perform the same process to simplify this equation. Computing the difference between $\Delta x_1(u+h)$ and $\Delta x_1(u)$ as $\Delta x_2(u)$, we get

$$\begin{aligned}
\Delta x_2(u) &= \Delta x_1(u+h) - \Delta x_1(u) \\
&= (3ah)(u+h)^2 + (3ah^2 + 2bh)(u+h) + (ah^3 + bh^2 + ch) \\
&\quad - [(3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch)] \\
&= 3ahu^2 + 6ah^2u + 3ah^3 + (3ah^2 + 2bh)u + 3ah^3 + 2bh^2 \\
&\quad + (ah^3 + bh^2 + ch) - [(3ah)u^2 + (3ah^2 + 2bh)u + (ah^3 + bh^2 + ch)] \\
&= 6ah^2u + (6ah^3 + 2bh^2)
\end{aligned}$$

This changes our pseudocode to the following:

```
u = 0;
x = d;
output(x);
dx1 = ah^3 + bh^2 + ch;
dx2 = 6ah^3 + 2bh^2;
for ( i = 1; i <= n; i++)
{
    u += h;
    x += dx1;
    output(x);
    dx1 += dx2;
    dx2 = 6ah^2u + (6ah^3 + 2bh^2);
}
```

We can carry this one final step further to remove the linear equation for Δx_2. Computing the difference between $\Delta x_2(u+h)$ and $\Delta x_2(u)$ as $\Delta x_3(u)$, we get

$$\begin{aligned}\Delta x_3(u) &= \Delta x_2(u+h) - \Delta x_2(u) \\ &= 6ah^2(u+h) + (6ah^3 + 2bh^2) \\ &\quad - 6ah^2u + (6ah^3 + 2bh^2) \\ &= 6ah^2u + 6ah^3 + (6ah^3 + 2bh^2) \\ &\quad - 6ah^2u + (6ah^3 + 2bh^2) \\ &= 6ah^3\end{aligned}$$

Our final code for forward differencing becomes the following:

```
x = d;
output(x);
dx1 = ah^3 + bh^2 + ch;
dx2 = 6ah^3 + 2bh^2;
dx3 = 6ah^3;
for ( i = 1; i <=n; i++ )
{
    x += dx1;
    output(x);
    dx1 += dx2;
    dx2 += dx3;
}
```

We have simplified our evaluation of x from 3 multiplies and 3 adds, down to 3 adds. We'll have to perform similar calculations for y and z, with differing deltas and a, b, c, and d values for each coordinate, giving a total of 9 adds for each point.

Note that forward differencing is only possible if the time steps between each point are equal. Because of this, in general we can't use it for animating along a curve, as time between frames may vary from frame to frame. In this case, the appropriate Horner's rule for our degree of polynomial is the most efficient solution.

10.4.2 Midpoint Subdivision

An alternative method for generating points along a curve is to recursively subdivide the curve until we have a set of subcurves, each of which can be approximated by a line segment. This subdivision is usually set to stop at a certain resolution that depends on our needs. This may end up with a

more accurate and more efficient representation of the curve than forward differencing, since more curve segments will be generated in areas with high curvature (areas that we might cut across with forward differencing) and fewer in areas with lower curvature.

We can perform this subdivision by taking a curve $Q(u)$ and breaking it into two new curves $L(s)$ and $R(t)$, usually at the midpoint $Q(1/2)$. In this case, $L(s)$ is the subcurve of $Q(u)$ where $0 \leq u \leq 1/2$, and $R(t)$ is the subcurve where $1/2 \leq u \leq 1$. The parameters s and t are related to u by

$$s = 2u$$

$$t = 2u - 1$$

Each subcurve is then tested for relative "straightness"—if it can be approximated well by a line segment, we stop subdividing, otherwise we keep going. The general algorithm looks like the following:

```
void
RenderCurve( Q )
{
    if ( Straight( Q ) )
        DrawLine( Q(0), Q(1) );
    else
    {
        MidpointSubdivide( Q, &L, &R );
        RenderCurve( L );
        RenderCurve( R );
    }
}
```

There are a few ways of testing how straight a curve is. The most accurate is to measure the length of the curve and compare it to the length of the line segment between the curve's two extreme points. If the two lengths are within a certain tolerance ϵ, then we can say the curve is relatively straight. This assumes that we have an efficient method for computing the arc length of a curve. We discuss some ways of calculating this in the next section.

Another method is to use the two endpoints and the midpoint (Figure 10.23(a)). If the distance between the midpoint and the line segment formed by the two endpoints is close to 0, then we can usually say that the curve is relatively close to a line segment. The one exception is when the curve crosses the line segment between the two endpoints (Figure 10.23(b)), which will result in a false positive when clearly the curve is not straight. To avoid the worst examples of this case, Parent [88] recommends performing forward differencing down to a certain level and only then adaptively subdividing.

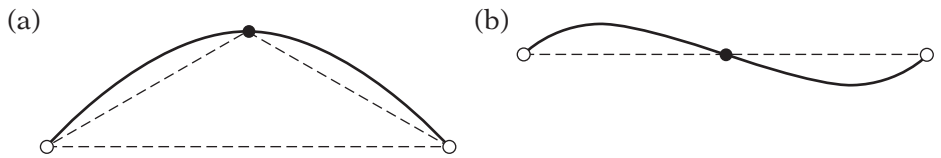


FIGURE 10.23 Midpoint test for curve straightness. (a) Total distance from endpoints to midpoint (black dot) is compared to distance between endpoints, (b) example of midpoint test failure.

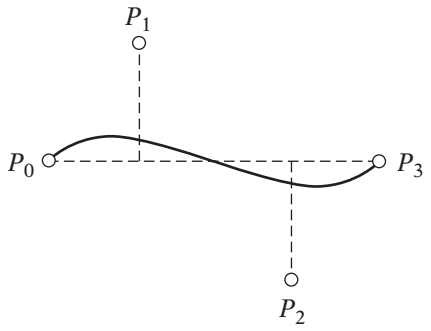


FIGURE 10.24 Test of straightness for Bézier curve. Measure distance of P_1 and P_2 to line segment P_0P_3.

The convex hull properties of the Bézier curve lead to a particularly efficient method for testing straightness, with no need to calculate a midpoint. If the interior control points are incident with the line segment formed by the two exterior control points, the area of the convex hull is 0, and the curve generated is itself a line segment. So for a cubic Bézier curve, we can test distance squared between the line segment formed by P_0 and P_3 and the two control points P_1 and P_2 (Figure 10.24). If both squared distances are less than some tolerance value, then we can say that the curve is relatively straight.

How we subdivide the curve if it fails the test depends on the type of curve. The simplest curves to subdivide are Bézier curves. To achieve this, we will generate new control points for each subcurve from our existing control points. So for a cubic curve, we will compute new control points L_1, L_2, L_3, and L_4 for curve L, and new control points R_1, R_2, R_3, and R_4 for curve R. These can be built by using a technique devised by de Casteljau. This method — known as *de Casteljau's method* — geometrically evaluates a Bézier curve at a given parameter u, and as a side effect creates the new control points needed to subdivide the curve at that point.

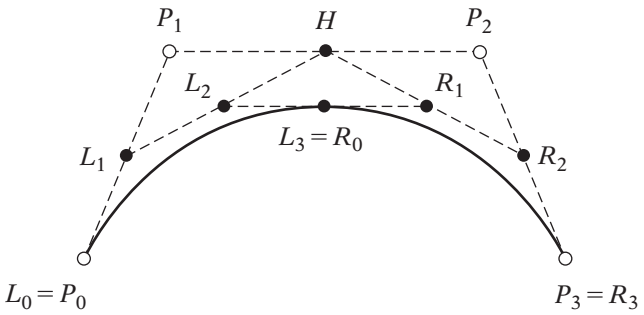


FIGURE 10.25 de Casteljau's method for subdividing Bézier curves.

Figure 10.25 shows the construction for a cubic Bézier curve. L_0 and R_3 are already known: They are the original control points P_0 and P_3, respectively. Point L_1 lies on segment $\overline{P_0P_1}$ at position $(1-u)P_0 + uP_1$. Similarly, point H lies on segment $\overline{P_1P_2}$ at $(1-u)P_1 + uP_2$, and point R_2 at $(1-u)P_2 + uP_3$. We then linearly interpolate along the newly formed line segments $\overline{L_1H}$ and $\overline{HR_2}$ to form $L_2 = (1-u)L_1 + uH$ and $R_1 = (1-u)H + uR_2$. Finally, we split segment $\overline{L_2R_1}$ to find $Q(u) = L_3 = R_1 = (1-u)L_2 + uR_1$.

Using the midpoint to subdivide is particularly efficient in this case. It takes only 6 adds and 6 multiplies (to perform the division by 2).

```
L0 = P0;
R3 = P3;
L1 = (P0 + P1)*0.5f;
 H = (P1 + P2)*0.5f;
R2 = (P2 + P3)*0.5f;
L2 = (L1 + H)*0.5f;
R1 = (H + R2)*0.5f;
L3 = R0 = (L2 + R1)*0.5f;
```

Subdividing other types of curves, in particular B-splines, can be handled by using an extension of this method devised by Boehm [12]. More information on Boehm subdivision and knot insertion can be found in Bartels et al. [6].

10.4.3 Computing Arc Length

We can informally define the *arc length* s between points $Q(u_1)$ and $Q(u_2)$ on a continuous curve Q as the distance along the curve between those two

points. At first glance, computing the length of a curve may not appear to be very related to sampling and tessellation. However, as mentioned above, some methods for subdividing a curve require knowing the arc lengths of subsections of that curve. Also, as we'll see, some arc length methods require sampling the curve to obtain their results.

The most accurate method of computing the length of a smooth curve (see Appendix B on the CD-ROM) $Q(u)$ from $Q(a)$ to $Q(b)$ is to directly compute the line integral:

$$s = \int_a^b \left\| Q'(u) \right\| du$$

Unfortunately, for most cubic polynomial curves, it is not possible to find an analytic solution to this integration. For quadratic curves, there is a closed-form solution, but evaluating the resulting functions is more expensive than using a numerical method that gives similar accuracy. In any case, if we wish to vary our curve types, we would have to redo the calculation and so it is not always practical.

The usual approach is to use a numerical method to solve the integral. There are many methods, which Burden and Faires [14] cover in some detail. In this case, the most efficient for its accuracy is Gaussian quadrature, since it attempts to minimize the number of function evaluations, which can be expensive. It approximates a definite integral from -1 to 1 by a weighted sum of unevenly spaced function evaluations, or

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} c_i f(x_i)$$

The actual c_i and x_i values depend on n and are carefully selected to give the best approximation to the integral. Appendix B (found on the CD-ROM) tabulates values up to $n = 5$, and Burden and Faires [14] describe in detail how these are derived for arbitrary values of n.

The restriction that we have to integrate over $[-1, 1]$ is not a serious obstacle. For a general definite integral over $[a, b]$, we can remap to $[-1, 1]$ by

$$\int_a^b f(x)dx = \int_{-1}^{1} f\left(\frac{(b-a)t + b + a}{2}\right) \frac{b-a}{2} dt$$

Guenter and Parent [50] describe a method that uses Gaussian quadrature in combination with adaptive subdivision to get very efficient results when

computing arc length. Similar to using adaptive subdivision for rendering, we cut the current curve segment in half. We use Gaussian quadrature to measure the length of each half, and compare their sum to the length of the entire curve, again computed using Gaussian quadrature. If the results are close enough, we stop and return the sum of lengths of the two halves. Otherwise, we recursively compute their lengths via subdivision.

There are other arc length methods that don't involve computing the integral in this manner. One is to subdivide the curve and use the sums of the lengths of the line segments created to approximate arc lengths at each of the subdivision points. We can create a sorted table of pairs (u_i, s_i), where u_i is the parameter for each subdivision, and s_i is the corresponding length at the point $Q(u_i)$. Since both u and *len* are monotonically increasing, we can sort by either parameter. An example of such a table can be seen in Table 10.1.

To find the length from the start of the curve for a given u, we search through the table to find the two neighboring entries with parameters u_k and u_{k+1} such that $u_k \leq u \leq u_{k+1}$. Since the entries are sorted, this can be handled efficiently by a binary search. The length then can be approximated by linearly interpolating between the two entries:

$$s \approx \frac{u_{k+1} - u}{u_{k+1} - u_k} s_k + \frac{u - u_k}{u_{k+1} - u_k} s_{k+1}$$

A higher-order curve can be used to get a better approximation.

To find the length between two parameters a and b where $a \leq b$, we compute the length for each and subtract one from the other, or

$$\text{length}(Q, a, b) = \text{length}(Q, b) - \text{length}(Q, a)$$

TABLE 10.1 Mapping parameter value to arc length

u	s
0.0	0.0
0.1	0.2
0.15	0.3
0.29	0.7
0.35	0.9
0.56	1.1
0.72	1.6
0.89	1.8
1.00	1.9

If we are using cubic Bézier curves, we can use a method described by Gravesen [47]. First of all, given a parameter u we can subdivide the curve (using de Casteljau's method) to be the subcurve from $[0, u]$. The new control points for this new subcurve can be used to calculate bounds on the length. The length of the curve is bounded by the length of the chord $\overline{P_0P_3}$ as the minimum, and the sum of the lengths of the line segments $\overline{P_0P_1}$, $\overline{P_1P_2}$, and $\overline{P_2P_3}$ as the maximum. We can approximate the arc length by the average of the two, or

$$\begin{aligned} L_{min} &= \|P_3 - P_0\| \\ L_{max} &= \|P_1 - P_0\| + \|P_2 - P_1\| + \|P_3 - P_2\| \\ L &\approx \frac{1}{2}(L_{min} + L_{max}) \end{aligned}$$

The error can be estimated by the square of the difference between the minimum and maximum:

$$\xi = (L_{max} - L_{min})^2$$

If the error is judged to be too large, then the curve can be subdivided and the length becomes the sum of the lengths of the two halves. Gravesen [47] states that for m subdivisions the error drops to 0 as 2^{-4m}.

A final alternative is presented by Vincent and Forsey [115]. Their method notes that for three neighboring curve points P_0, P_1, and P_2, the length of arc through them can be approximated by $D_2 + (D_2 - D_1)/3$, where

$$\begin{aligned} D_1 &= \|\overline{P_0P_2}\| \\ D_2 &= \|\overline{P_0P_1}\| + \|\overline{P_1P_2}\| \end{aligned}$$

This assumes that P_1 is relatively equidistant from P_0 and P_2, and the arc has low curvature. To improve accuracy, estimates for a given segment (say $\overline{P_1P_2}$) can be computed using the neighboring point on either side (i.e., in our case, one estimate is computed using P_0 and another using P_3). These are averaged to give the final result.

The general algorithm begins by sampling an odd number of points across the curve. For each interior segment, it determines if more samples need to be taken due to high curvature. If so, it computes those recursively until a low enough curvature is reached. Otherwise, it computes the segment length and returns. For general curves, this technique is slightly more expensive than Gaussian quadrature, but it does handle certain pathological cases (such as cusps) better.

10.5 Controlling Speed along a Curve

10.5.1 Moving at Constant Speed

SOURCE CODE DEMO SpeedControl

One common requirement for animation is that the object animated move at a constant speed along a curve. However, in most interesting cases, using a given curve directly will not achieve this. The problem is that in order to achieve variety in curvature, the first derivative must vary as well, and hence the distance we travel in a constant time will vary depending on where we start on the curve. For example, Figure 10.26 shows a curve subdivided at equal intervals of the parameter u. The lengths of the subcurves generated vary greatly from one to another.

Ideally, given a constant rate of travel r and time of travel t, we'll want to cover a distance of $s = rt$. So given a starting parameter u_1 on the curve, we want to find the parameter u_2 such that the arc length between $Q(u_1)$ and $Q(u_2)$ equals s.

We've already discussed a number of methods for computing the arc length of a curve. Regardless of the method we use, we'll assume we have some function $G(u)$ that returns the length s from $Q(0)$ to $Q(u)$. So, for the case where $u_1 = 0$, we can use the inverse function $G^{-1}(s)$ to determine the parameter u_2, given an input length s. This is known as a reparameterization by arc length. Unfortunately, in general the arc length function for a parameterized curve is impossible to invert in terms of a finite number of elementary functions, so numerical methods are used instead.

One way is to note that finding u_2 is equivalent to the problem of finding the solution u of the equation

$$s - \text{length}(u_1, u) = 0 \tag{10.18}$$

A method that allows us to solve this is Newton-Raphson root finding. Burden and Faires [14] present a derivation for this using the Taylor series expansion.

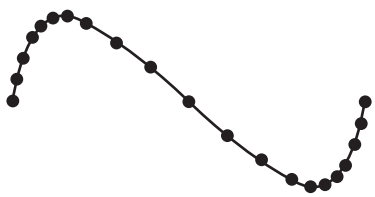


Figure 10.26 Parameter-based subdivision of curve, showing non-equal segment lengths.

Suppose we have a function $f(x)$ where we want to find p such that $f(p) = 0$. We begin with a guess for p, which we'll call $\bar{x}$, such that $f'(\bar{x}) \neq 0$ and $|p - \bar{x}|$ is relatively small. In other words, $\bar{x}$ may not quite be p but it's a pretty good guess. If we use $\bar{x}$ as a basis for the Taylor series polynomial, we get

$$f(x) = f(\bar{x}) + (x - \bar{x}) f'(\bar{x}) + \frac{1}{2}(x - \bar{x})^2 f''(\xi(x))$$

We assume that $\xi(x)$ is bounded by x and $\bar{x}$, so we can ignore the remainder of the terms. If we substitute p for x, then $f(p) = 0$ and

$$0 = f(\bar{x}) + (p - \bar{x}) f'(\bar{x}) + \frac{1}{2}(p - \bar{x})^2 f''(\xi(x))$$

Since $|p - \bar{x}|$ is relatively small, we assume that $(p - \bar{x})^2$ is small enough that we can ignore it, and so

$$0 \approx f(\bar{x}) + (p - \bar{x}) f'(x)$$

Solving for p gives us

$$p \approx \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})} \tag{10.19}$$

This gives us our method. We make an initial guess $\bar{x}$ at the solution and use the result of equation 10.19 to get a more accurate result p. If p still isn't close enough, then we feed it back into the equation as $\bar{x}$ to get a still more accurate result, and so on until we reach a solution of sufficient accuracy or after a given number of iterations is performed.

For our initial guess in solving equation 10.18, Eberly [25] recommends taking the ratio of our traveled length to the total arc length of the curve and mapping it to our parameter space. Assuming our curve is normalized so that u is in $[0, 1]$, then pseudocode for our root-finding method will look like the following:

```
float FindParameterByDistance( float u1, float s )
{
    // ensure that we remain within valid parameter space
    if (s > ArcLength(u1,1.0f))
        return 1.0f;

    // get total length of curve
    float len = ArcLength(0.0f,1.0f);
```

```
        // make first guess
        float p = u1 + s/len;

        for (int i = 0; i < MAX_ITER; ++i)
        {
            // compute function value and test against zero
            float func = ArcLength(u1,p) - s;
            if ( fabsf(func) < EPSILON )
            {
                return p;
            }

             // perform Newton-Raphson iteration step
            p -= func/Length(Derivative(p));
        }

        // done iterating, return last guess
        return p;
    }
```

The first test ensures that the distance we wish to travel is not greater than the remaining length of the curve. In this case, we assume that this is the last segment of a piecewise curve and just jump to the end. A more robust implementation should subtract the remaining length from the distance and restart at the beginning of the next segment.

Computing the derivative of the curve is simple, as this is easily derived from the definition of the curve, as we did for clamped and natural splines. However, there is a serious problem if `Length(Derivative(p))` is zero or near zero. This will lead to a division by zero and we will end up subtracting `NaN` from `p`, which will give us a garbage result.

The solution is to use an alternative root-finding technique known as bisection. It makes use of the mean value theorem, which states that if you have a function $f(x)$ that's continuous on $[a, b]$ and $f(a)f(b) < 0$ (i.e., $f(a)$ and $f(b)$ have opposite signs), then there is some value p between a and b where $f(p) = 0$. This is definitely true in our case. The length of the curve is monotonically increasing, so there will be only one zero. If it's not at the beginning, then $f(a) = length(a) - s = -s$, which is less than zero. If it's not at the end, then $f(b) = length(b) - s$, which is greater than zero. Our endpoints have differing signs, so we can use the bisection method.

This is solved by doing a binary search: We cut the problem interval in two and search further in the more fruitful half. The problem with bisection is that it converges considerably slower than Newton-Raphson, so sometimes we want to use Netwon-Raphson and sometimes bisection. Therefore, our

hybrid approach will look like (for brevity's sake we have only included the parts that are different) the following:

```
float FindParameterByDistance( float u1, float s )
{
    // set endpoints for bisection
    float a = u1;
    float b = 1.0f;

    // ensure that we remain within valid parameter space

    // get total length of curve

    // make first guess

    for (int i = 0; i < MAX_ITER; ++i)
    {
        // compute function value and test against zero

        // update endpoints for bisection
        if (func < 0.0f)
            a = p;
        else
            b = p;

        // compute speed
        speed = Length(Derivative(p));
        if (bisection)
            // do bisection step
            p = 0.5f*(a+b);
        else
            // perform Newton-Raphson iteration step
            p -= func/speed;
    }

    // done iterating, return last guess
    return p;
}
```

The only remaining question is how we determine to use bisection over Newton-Raphson. One obvious possibility is to check whether the speed is zero, as that got us into trouble in the first place. However, that's not

enough. If the speed is nonzero but sufficiently small, `func/speed` could be sufficiently large to cause us to step outside the bisection interval or even the valid parameter space of the curve. So that gives us our test: If `p - func/speed` is less than `a` or greater than `b`, use bisection. We can write this as follows.

```
if (p - func/speed < a || p - func/speed > b)
    // do bisection step
else
    // perform Newton-Raphson iteration step
```

Multiplying by `speed` and rearranging terms gives us the following:

```
if ((p-a)*speed < func || (p-b)*speed > func)
    // do bisection step
else
    // perform Newton-Raphson iteration step
```

Press et al. [96] further recommend the following so as to be floating-point safe:

```
if (((p-a)*speed - func)*((p-b)*speed - func) > 0.0f)
    // do bisection step
else
    // perform Newton-Raphson iteration step
```

That should solve our problem.

A few other implementation notes are in order at this point. As we've seen, computing `ArcLength()` can be a nontrivial operation. Because of this, if we're going to be calling `FindParameterByDistance()` many times for a fixed curve, it is more efficient to precompute `ArcLength(0.0f, 1.0f)` and use this stored value instead of recomputing it each time. Also, the constants `MAX_ITER` and `EPSILON` will need to be tuned depending on the type of curve and the number of iterations we can feasibly calculate due to performance constraints. Reasonable starting values for this tuning process are 32 for `MAX_ITER` and 1.0e-06f for `EPSILON`.

As a final note, there is an alternative approach if we've used the table-driven method for computing arc length. Recall that we used Table 10.1 to compute s given a parameter u. In this case, we invert the process and search for the two neighboring entries with lengths s_j and s_{j+1} such that $s_j \leq s \leq s_{j+1}$.

Again, we can use linear interpolation to approximate the parameter u, which gives us length s as

$$u \approx \frac{s_{j+1} - s}{s_{j+1} - s_j} u_j + \frac{s - s_j}{s_{j+1} - s_j} u_{j+1}$$

To find the parameter b given a starting parameter a and a length s, we compute the length at a and add that to s. We then use the preceding process with the total length to find parameter b.

The obvious disadvantage of this scheme is that it takes additional memory for each curve. However, it is simple to implement, somewhat fast, and does avoid the Newton-Raphson iteration needed with other methods.

10.5.2 Moving at Variable Speed

In our original equation for computing the desired distance to travel, $s = rt$, we assumed that we were traveling at a constant rate of speed. However, it is often convenient to have an adjustable rate of speed over the length of the curve. We can represent this by a general distance–time function $s(t)$, which maps a time value t to the total distance traveled from t_0. As an example, Figure 10.27 shows $s(t) = rt$ as a distance–time graph.

Other than traveling at a constant rate, the most common distance–time function is known as *ease-in/ease-out*. Here, we start at a zero rate of speed, accelerate up to a constant nonzero rate of speed in the middle, and then decelerate down again to a stop. This feels natural, as it approximates the

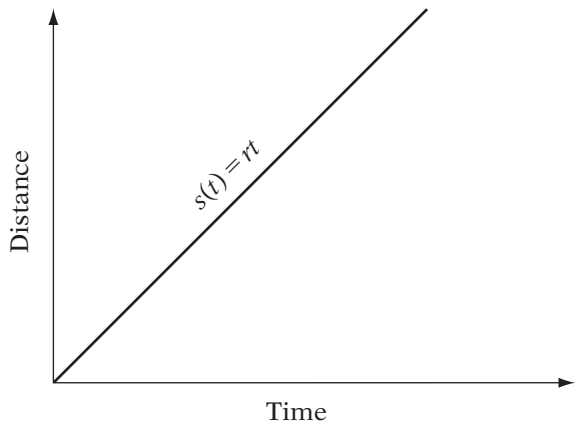


Figure 10.27 Example of distance–time graph: moving at constant speed.

need to accelerate a physical camera, move it, and slow it down to a stop. Figure 10.28 shows the distance–time graph for one such function.

Parent [88] describes two methods for constructing ease-in/ease-out distance–time functions. One is to use sinusoidal pieces for the acceleration/deceleration areas of the function and a constant velocity in the middle. The pieces are carefully chosen to ensure C^1 continuity over the entire function. The second method involves setting a maximum velocity that we wish to attain in the center part of the function and assumes that we move with constant acceleration in the opening and closing ease-in/ease-out areas. This gives a velocity–time curve as in Figure 10.29. By integrating this, we get a distance–time curve. By assuming that we start at the beginning of the curve, this gives us a piecewise curve with parabolic acceleration and deceleration.

However, there is no reason to stop with an ease-in/ease-out distance–time function. We can define any curve we want, as long as the curve remains within the positive d and t axes for the valid time and distance intervals. One possibility is to let the user trace out a curve, but that can lead to invalid inputs and difficulty of control. Instead, animation packages such as those in 3D Studio Max and Maya allow artists to create these curves by setting keys with

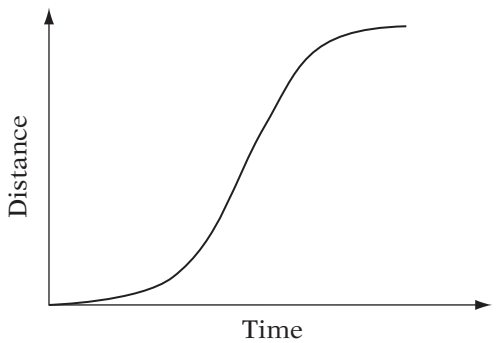


Figure 10.28 Example of distance–time graph: Ease-in/ease-out function.

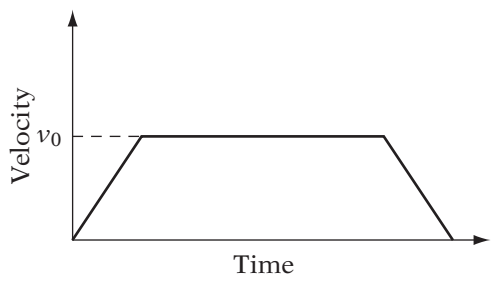


Figure 10.29 Example of velocity–time function: Ease-in/ease-out with constant acceleration/deceleration.

particular arrival and departure characteristics. Standard parlance includes such terms as fast-in, fast-out, slow-in, and slow-out. *In* and *out* in this case refer to the incoming and outgoing speed at the key point, respectively; *fast* means that the speed is greater than 1, and *slow* that it is less than 1. An example curve with both fast-in/fast-out and slow-in/slow-out can be seen in Figure 10.30. There also can be linear keys, which represent the linear rate seen in Figure 10.27, and step-keys, where distance remains constant for a certain period of time and then abruptly changes, as in Figure 10.31. Alternatively, the user may specify no speed characteristics and just expect the program to pick an appropriately smooth curve.

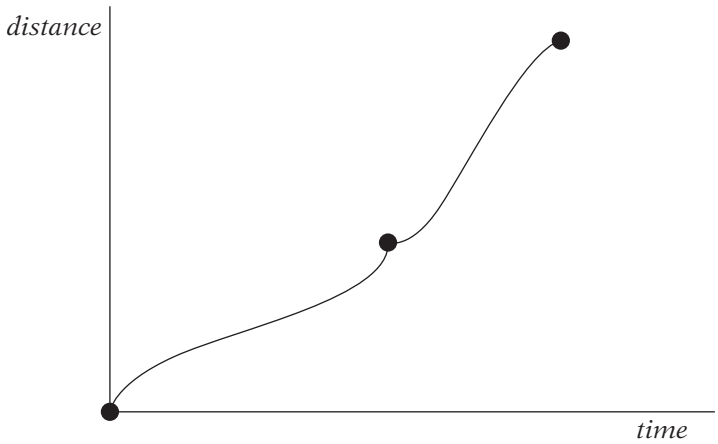


FIGURE 10.30 Example of distance–time graph: Fast-out/fast-in followed by slow-out/slow-in.

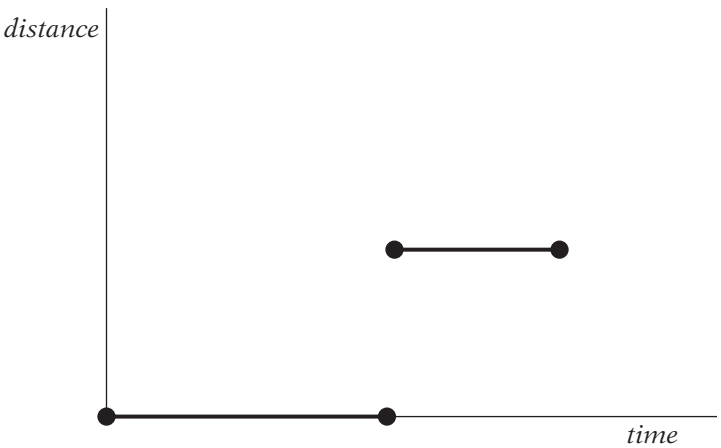


FIGURE 10.31 Example of distance–time graph: Step-key transition.

With all of these, the final distance–time curve can be easily generated with the techniques described in Section 10.2.3. More detail can be found in Van Verth [114].

10.6 Camera Control

One common use for a parametric curve is as a path for controlling the motion of a virtual camera. In games this comes into play most often when setting up in-game cinematics, where we want to play a series of scripted events in engine while giving a game a cinematic feel via the clever use of camera control. For example, we might want to have a camera track around a pair of characters as they dance about a room. Or, we might want to simulate a crane shot zooming from a far point of view right down into a close-up. While either of these could be done programmatically, it would be better to provide external control to the artist, who most likely will be setting up the shot. The artist sets the path for the camera — all the programmer needs to do is provide code to move the camera along the given path.

Determining the position of the camera isn't a problem. Given the start time t_s for the camera and the current time t_c, we compute the parameter $t = t_c - t_s$ and then use our time controls together with our curve description to determine the current position at $Q(t)$.

Computing orientation is another matter. The most basic option is to set a fixed orientation for the entire path. This might be appropriate if we are trying to create the effect of a panning shot but is rather limiting and somewhat static. Another way would be to set orientations at each sample time as well as positions, and interpolate both. However, this can be quite time consuming and may require more keys to get the effect we want.

A further possibility is to use the Frenet frame for the curve. This is an orthonormal frame with an origin of the current position on the curve, and a basis $\{\widehat{\mathbf{T}}, \widehat{\mathbf{N}}, \widehat{\mathbf{B}}\}$ where $\widehat{\mathbf{T}}$ points in the direction of the first derivative, $\widehat{\mathbf{N}}$ points roughly in the direction of the second derivative, and $\widehat{\mathbf{B}}$ is the cross product of the first two. The vector $\widehat{\mathbf{T}}$ acts as our view direction vector, $\widehat{\mathbf{N}}$ acts as our view side vector, and $\widehat{\mathbf{B}}$ acts as our view up vector.

For any curve specified by the matrix form $Q(u) = \mathbf{UMG}$, we can easily compute the first derivative by using the form $\mathbf{Q}'(u) = \mathbf{U}'\mathbf{MG}$, where for a cubic curve

$$\mathbf{U}' = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix}$$

Similarly, we can compute the second derivative as $\mathbf{Q}''(u) = \mathbf{U}''\mathbf{MG}$ where

$$\mathbf{U}'' = \begin{bmatrix} 6u & 2 & 0 & 0 \end{bmatrix}$$

As mentioned, we set $\mathbf{T} = \mathbf{Q}'(u)$. We compute $\mathbf{B}$ as the cross product of the first and second derivatives:

$$\mathbf{B} = \mathbf{Q}'(u) \times \mathbf{Q}''(u)$$

Then, finally, $\mathbf{N}$ is the cross product of the other two:

$$\mathbf{N} = \mathbf{B} \times \mathbf{T}$$

Normalizing $\mathbf{T}$, $\mathbf{N}$, and $\mathbf{B}$ gives us our orthonormal basis.

Parent [88] describes a few flaws with using the Frenet frame directly. First of all, the second derivative may be $\mathbf{0}$, which means that $\widehat{\mathbf{B}}$ and hence $\widehat{\mathbf{N}}$ will be $\mathbf{0}$. One solution is to interpolate between two frames on either side of our current location. Since the second derivative is zero, or near zero, the first derivative won't be changing much, so we're really interpolating between two frames in $\mathbb{R}^2$. This consists of finding the angle between them and interpolating along that angle (Figure 10.32). The one flaw with this is that when finding these frames we're still using $\mathbf{Q}''$, which may be near zero and hence lead to floating-point issues. In particular, if we are moving with linear motion, there will be no valid neighboring values for estimating $\mathbf{Q}''$.

Then, too, it assumes that the second derivative exists for all values of t, namely, that $Q(t)$ is C^2 continuous. Many of the curves we've discussed, in particular the piecewise curves, do not meet this criterion. In such cases, the camera will rather jarringly change orientation. For example, suppose we have two curve segments as seen in Figure 10.33, where the second derivative instantly changes to the opposite direction at the join between the segments. In the Frenet frame for the first segment, the $\mathbf{w}$ vector points out of the page. In the second segment, it points into the page. As the camera crosses the join, it will instantaneously flip upside down. This is probably not what the animator had in mind.

Finally, we may not want to use the second derivative at all. For example, if we have a path that heads up and then down, like a hill on a roller coaster,

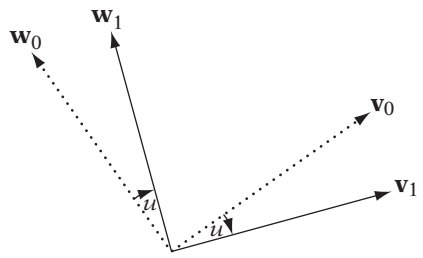


FIGURE 10.32 Interpolating between two path frames.

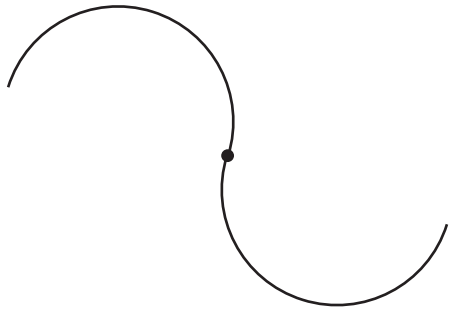


FIGURE 10.33 Frame interpolation issues. Discontinuity of second derivative at point.

the direction of the second derivative points generally down along that section of path. This means that our view up vector will end up parallel to the ground for that section of curve—again, probably not the intention of the animator.

A further refinement of this technique is to use something called the *parallel transport frame* [53]. This is an extension of the interpolation technique shown in Figure 10.32. We begin at a position with a valid frame. At the next time step, we compute the derivative, which gives us our view direction vector as before. To compute the other two vectors, we rotate the previous frame by the angle between the current derivative and the previous derivative. If the vectors are parallel, we won't rotate at all, which solves the problem where the second derivative may be zero. This will generate a smooth transition in orientation across the entire path, but doesn't provide much control over expected behavior, other than setting the initial orientation.

An alternative solution is to adopt a technique from Chapter 6. Again we use the first derivative as our view direction vector, but instead generate the view up vector from this and the world up vector. The view side vector is the cross product of these two. This solves the problem, but does mean that if we have a fixed up vector we can't roll our camera through a banking turn—its up vector will remain relatively aligned with the given up vector.

A refinement of this is to allow user-specified up vectors at each sample position, which default to the world up vector. The program would interpolate between these up vectors just as it interpolates between the positions. Alternatively, the user could set a path $U(t)$ that is used to calculate the up vector: $\mathbf{v}_{up} = U(t) - Q(t)$. The danger here is that the user may specify two up vectors of opposing directions that end up interpolating to $\mathbf{0}$, or an up vector that aligns with the view direction vector, which would lead to a cross product of $\mathbf{0}$. If the user is allowed this kind of flexibility, recovery cases and some sort of error message will be needed.

We can take this one step further by separating our view direction from the Frenet frame and using our familar look-at point method, again from Chapter 6. The choice of what we use as our look-at point can depend on the camera effect desired. For example, we might pick a fixed point on the ground and then perform a fly-by. We could use the position of an object or the centroid of positions for a set of objects. We could set an additional path, and use the position along that path at our current time, to give the effect of a moving point of view without tying it to a particular object.

Another possibility is to look ahead along our current path a few steps in time, as if we were following an object a few seconds ahead of us. So, if we're at position $Q(t)$, we use as our look-at point the position $Q(t + \delta t)$. In this situation, we have to be sure to reparameterize the curve based on arc length, because otherwise the distance $\|Q(t) - Q(t + \delta t)\|$ may change depending on where we are on the curve, which may lead to odd changes in the view direction.

An issue with this technique is that it may make the camera seem clairvoyant, which can ruin the drama in some situations. Also, if our curve is particularly twisty, looking ahead may lead to sudden changes in direction. We can smooth this by averaging a set of points ahead of our position on the curve. How separated the points are makes a difference: too separated and our view direction may not change much; too close together and the smoothing effect will be nullified. It's usually best to make the amount of separation another setting available to the animator so that he or she can control the effect desired.

10.7 Chapter Summary

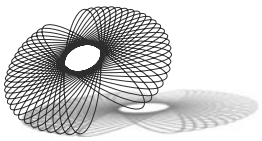
In this chapter we have touched on some of the issues involved with using parametric curves to aid in animation. We have discussed the most commonly used of the many possible curve types and how to subdivide these curves. Possible interfaces have been presented that allow animators and designers to create curves that can be used in the games they create. We have also covered some of the most common animation tasks beyond simple interpolation: controlling travel speed along curves and maintaining a logical camera orientation.

For rotations, fixed and Euler and axis–angle formats interpolate well only under simple circumstances. Matrices can be interpolated, but at significantly greater cost than quaternions. If you need to interpolate orientation, the clear choice is to use quaternions.

For further reading, Rogers and Adams [98] and Bartels et al. [6] present much of this material in greater detail, in particular focusing on B-splines. Parent [88] covers the use of splines in animation, as well as additional animation techniques. Burden and Faires [14] have a chapter on interpolation

and explain some of the numerical methods used with curves, in particular integration techniques and the Newton-Raphson method.

We have not discussed parametric surfaces, but many of the same principles apply: Surfaces are approximated or interpolated by a grid of points and are usually rendered using a subdivision method. Rogers [97] is an excellent resource for understanding how NURBS surfaces, the most commonly used parametric surfaces, are created and used.

Chapter 11

Random Numbers

11.1 Introduction

Now that we've spent some time in the deterministic worlds of pure mathematics, graphics, and interpolation, it's time to look at some techniques that can make our world look less structured and more organic. We'll begin in this chapter by considering randomness and generating random numbers in the computer.

So why do we need random numbers in games? We can break down our needs into a few categories: the basic randomness needed for games of chance, as in simulating cards and dice; randomness for generating behavior for intelligent agents such as enemies and nonplayer allies; turbulence and distortion for procedural textures; and randomly spreading particles, such as explosions and gunshots, in particle systems.

In this chapter we'll begin by covering some basic concepts in probability and statistics that will help us build our random processes. We'll then move to techniques for measuring random data and then basic algorithms for generating random numbers. Finally, we'll close by looking at some applications of our random number generators (RNGs).

11.2 Probability

Probability Theory is the mathematics of measuring the likelihood of unpredicable behavior. It was originally applied to games of chance such as dice and cards. In fact, Blaise Pascal and Pierre de Fermat worked out the basics of probability to solve a problem posed by a famous gambler, the

Chevalier de Mere. His question was: Which is more likely, rolling at least one 6 in 4 throws of a single die, or at least one double 6 in 24 throws of a pair of dice? (We'll answer this question at the end of the next section.)

These days probability can be used to predict the likelihood of other events such as the weather (i.e., the chance of rain is 60 percent) and even human behavior. In the following section we will summarize some elements of probability, enough for simple applications.

11.2.1 Basic Probability

The basis of probability is the *random experiment*, which is an experiment with a nondetermined outcome that can be observed and reobserved under the same conditions. Each time we run this experiment we call it a *random trial*, or just a trial. We call any of the particular outcomes of this experiment an *elementary outcome*, and the set of all elementary outcomes the *sample space*. Often we are interested in a particular set of outcomes, which we call the *favorable outcomes* or an *event*.

We define the *probability* of a particular event as a real number from 0 to 1, where 0 represents that the event will never happen, and 1 represents that the event will always happen. This value is also represented as a percentage, so from 0 percent to 100 percent. For a particular outcome ω_i, we can represent the probability as $P(\omega_i)$.

The classical computation of probability assumes that all outcomes are equally likely. In this case, the probability of an event is the number of favorable outcomes for that event divided by the total number of elementary outcomes. As an example, suppose we roll a fair (i.e., not loaded) six-sided die. This is our random experiment. The sample space Ω for our experiment is all the possible values on each side, so $\Omega = \{1, 2, 3, 4, 5, 6\}$. The event we're interested in is, how likely is it for a 3 or 4 to come up? Or, what is $P(3 \text{ or } 4)$? The number of favorable outcomes is 2 (either a 3 or a 4) and the number of all elementary outcomes is 6, so the probability is 2 over 6, or 1/3.

Another classic example is drawing a colored ball out of a jar. If we have 3 red balls, 2 blue balls, and 5 yellow balls, the probability of drawing a red ball out is $3/(3+2+5)$ or $3/10$, the probability of drawing a blue ball is $2/10 = 1/5$, and the probability of drawing a yellow ball is $5/10 = 1/2$.

However, it's not always the case that each outcome is equally likely (life is not necessarily fair). Because of this, there are two additional approaches to computing probabilities. The first is the frequentist approach, which has as its central tenet that if we perform a large number of trials, that the number of observed favorable outcomes over the number of trials will approach the probability of the event. This also is known as the *law of large numbers*. The

second is the Bayesian approach, which is more philosophical and is based on the fact that many events are not in practice repeatable. The probability of such events is based on a personal assessment of likelihood. Both have their applications, but for the purposes of this chapter, we will be focusing on the frequentist definition.

As an example of the law of large numbers, look at Figure 11.1. Figure 11.1(a) shows the result of a computer simulation of rolling a fair die 1,000 times. Each column represents the number of times each side came up. As we can see, while the columns are not equal, they are pretty close. For example, if we divide the number of 3s generated by the total number of rolls, we get 0.164 — pretty close to the actual answer of 1/6.

Figure 11.1(b), on the other hand, shows the result of rolling a loaded die, where 6s come up more often. As we'd expect, the 6 column is much higher than the rest, and dividing the number of 6s generated by the total gives us 0.286 — not at all close to the expected probability. Clearly something nefarious is going on. While we never can be exact about whether observed results match expected behavior (this is probability, after all), we'll talk later about a way to measure whether our observed outcomes match the expected outcomes.

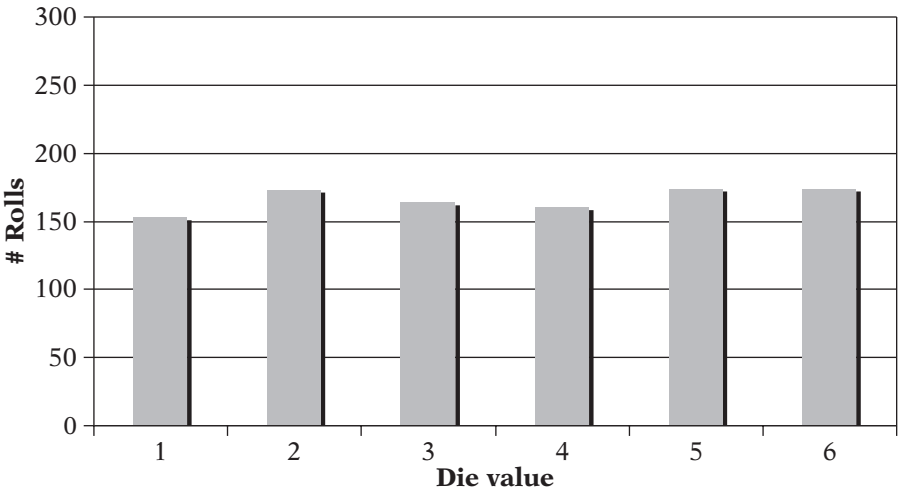
We often consider the probability of more than one trial at a time. If performing the experiment has no effect on the probability of future trials, we call these *independent events* or *independent trials*. For example, each instance of rolling a die is an independent trial. Drawing a ball out of the jar and not putting it back is not; future trials are affected by what happens. For example, if we draw a red ball out of the jar and don't replace it, the probability of drawing another red ball is 2/9, as there are now only 2 red balls and 9 balls total in the jar. These are known as *dependent events* or *dependent trials*.

A few algebraic rules for probability may prove useful for game development. First of all, the probability of an event *not* happening is 1 minus the probability of the event, or $P(\text{not } E) = 1 - P(E)$. For example, the probability of not rolling a 6 on a fair die is $1 - 1/6 = 5/6$.

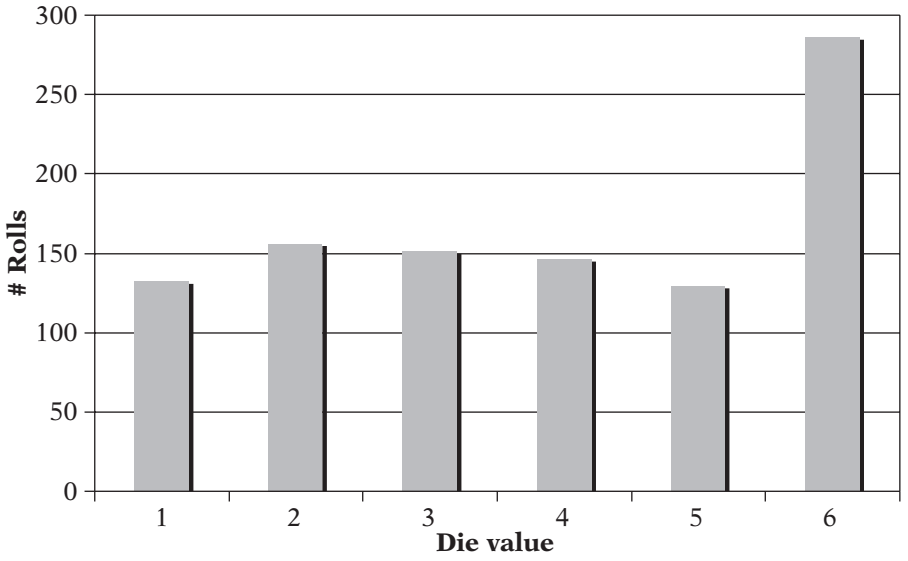
Secondly, the probability of two independent events E and F occuring is $P(E) \cdot P(F)$. So for example, the probability of rolling a die twice and rolling a 1 or 2 on the first roll and a 3, 4, or 6 on the second roll is $2/6 \cdot 3/6 = 6/36 = 1/6$.

Finally, the probability of one event E *or* another event F is $P(E) + P(F) - P(E \text{ and } F)$. An example of this is considering the probability of rolling an odd number or a 1 on a die. The probability of rolling an odd number *and* a 1 is just the probability of rolling a 1 or 1/6 (we can't use the multiplicative rule here because the events are not independent). So, the result is $3/6 + 1/6 - 1/6 = 1/2$ — just the probability of rolling an odd number.

With these rules we can answer Chevalier de Mere's question. The first part of the question is, what is the probability of rolling at least one 6 in 4



(a)



(b)

FIGURE 11.1 (a) Simulation results for rolling a fair die 1000 times, and (b) simulation results for rolling a loaded die 1000 times.

throws of a single die? We'll represent this as $P(E)$. It's a little easier to turn this around and ask, what is the probability of *not* rolling a 6 in 4 throws of one die? We can call the event of not throwing a 6 on the ith roll A_i, and the probability of this event is $P(A_i)$. Then the probability of all 4 is $P(A_1$ and A_2 and A_3 and $A_4)$. As each roll is an independent event, we can just multiply the 4 probabilities together to get a probability of $(5/6)^4$. But this probability is P(not E), so we must use the "not" rule and subtract the result from 1 to get $P(E) = 1 - (5/6)^4$ or 0.518.

The other half of the question is, what is the probability of rolling at least one double 6 in 24 throws of a pair of die? This can be answered similarly. We represent this as $P(F)$. Again, we turn the question around and compute the probability of the negative: rolling *no* double 6s. For a given roll i, the probability of not rolling a double 6 is $P(B_i) = 35/36$. We multiply the results together to get $P(\text{not } F) = (35/36)^{24}$ and so $P(F) = 1 - (35/36)^{24}$ or 0.491. So, the first event is more likely.

This is just a basic example of computing probabilities. Those interested in computing the probability of more complex examples are advised to look to the references noted at the end of the chapter — it can get more complicated than one expects, particularly when dealing with dependent trials.

11.2.2 Random Variables

As we saw with vectors, mathematicians like abstractions so they can wrap an algebra around a concept and perform symbolic operations on it. The abstraction in this case is the *random variable*. Suppose we have a random experiment that generates values (if not, we can assign a value to each outcome of our experiment). We call the values generated by this process a random variable, usually represented by X. Note that X represents all possible values; a particular result of a random experiment is represented by X_i, and a particular value is represented by x.

If the set of all random values for our given problem has a fixed size,[1] as in the examples above, then we say it is a *discrete random variable*. In this case, we're interested in the probability of a particular outcome x. We can represent this as a function $m(x)$, where the function's domain is the sample space Ω. As an example, suppose we create such a function for our jar experiment. We'll say that red = 1, blue = 2, and yellow = 3. The sample space of our random variable is now $\Omega = 1, 2,$ or 3. The value of $m(x)$ for each possible x is the probability that x is the result of the draw out of the jar. The resulting graph can be seen in Figure 11.2. Notice that $m(x)$ only has a value at 1, 2, or 3, and is 0 everywhere else. This is known as a *probability mass function*, or sometimes a

1. Or, is countably infinite, though in games this is rarely considered, if ever.

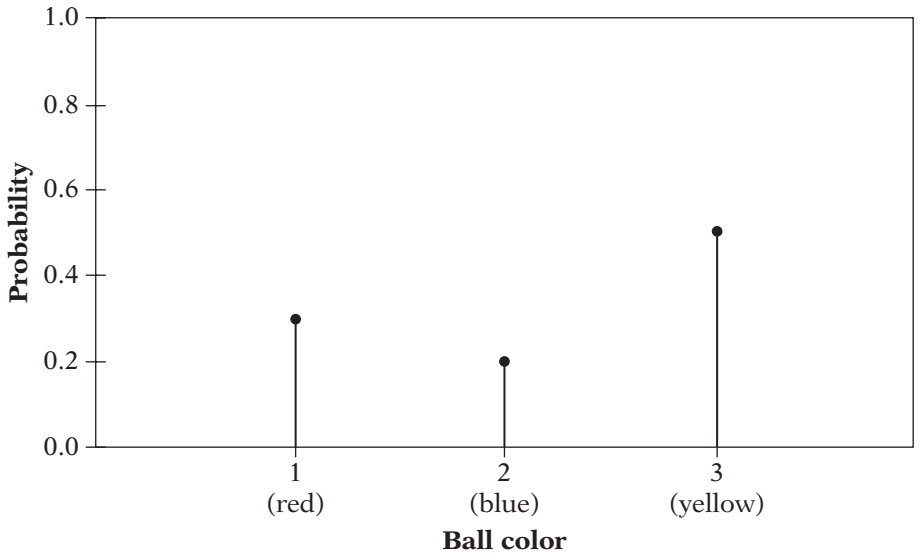


FIGURE 11.2 Probability mass function for drawing one ball out of a jar with 3 red balls, 2 blue balls and 5 yellow balls.

probability distribution function. This function has three important properties: its domain is the sample space of a random variable; for all values x, $m(x) \geq 0$ (i.e., there are no negative probabilities); and the sum of the probabilities of all outcomes is 1, or

$$\sum_{i=0}^{n-1} m(x_i) = 1$$

where n is the number of elements in Ω.

Now, suppose that our sample space has an uncountably infinite number of outcomes. One example of this is spinning a disc with a pointer: Its angle relative to a fixed mark has an infinite number of possible values. This is known as a *continuous random variable*. Another example of a continuous random variable is randomly choosing a value from all real values in the range [0, 1]. Assuming all numbers have an equal probability, this is known as a *uniform variate*, or sometimes as the *canonical random variable* ξ [91].

One interesting thing about a continuous random variable is that the probability of a given outcome x is 0, since the number of possible outcomes we're dividing by is infinite. However, we can still measure probabilities by considering ranges of values and use a special kind of function to

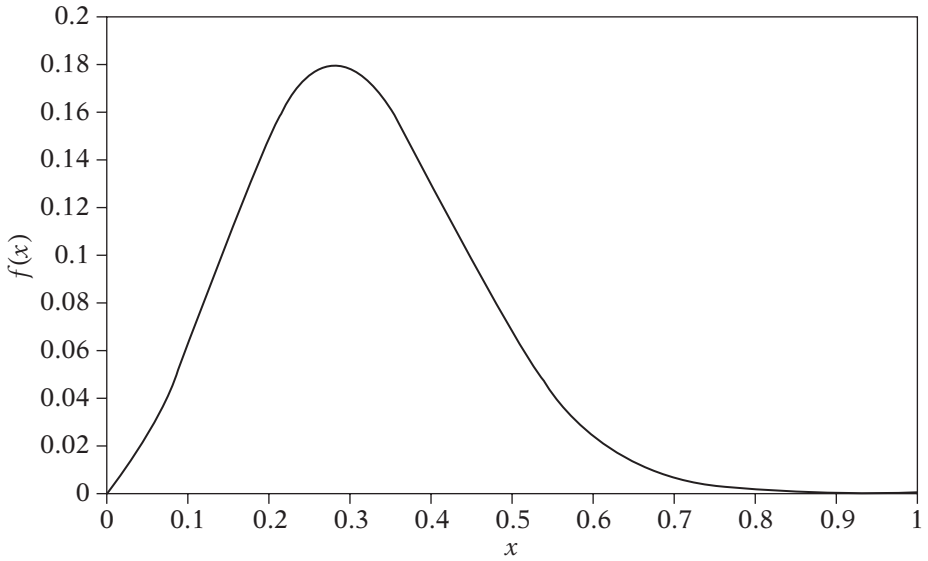


FIGURE 11.3 Example of a probability density function.

encapsulate this. Figure 11.3 shows one such function over the canonical random variable. This function $f(x)$ is known as a *probability density function* (PDF). It has characteristics similar to the probability mass function for the discrete case: All values $f(x) \geq 0$ and the area under the curve is equal to 1. As with the discrete case, the second characteristic indicates that the sum of the probabilities for all outcomes is 1 and can be represented by the integral:

$$\int_{-\infty}^{\infty} f(x)dx = 1$$

We can also find the probability of a series of random events, say from a to b. In the discrete case, all we need to do is take the sum across that interval:

$$P(a \leq \omega \leq b) = \sum_{x=a}^{b} m(x)$$

In the continuous case, again we take the integral:

$$P(a \leq x \leq b) = \int_{a}^{b} f(x)dx$$

Sometimes we want to know the probability of a random value being less than or equal to some value y. Using the mass function, we can compute this in the discrete case as

$$F(y) = \sum_{x=x_0}^{y} m(x)$$

or in the continuous case using the density function as

$$F(y) = \int_{-\infty}^{y} f(x)dx$$

This function $F(x)$ is known as the *cumulative distribution function* (CDF). We can think of this as a cumulative sum across the domain. Note that because the CDF is the integral of the PDF in the continuous realm, the PDF is actually the derivative of the CDF.

Figure 11.4 shows the cumulative distribution function for the continuous PDF in Figure 11.3. Note that it starts at a value of 0 for the minimum in the domain and increases to a maximum value of 1: All cumulative distribution functions have this property. We'll be making use of cumulative distribution functions when we discuss the chi-square method below.

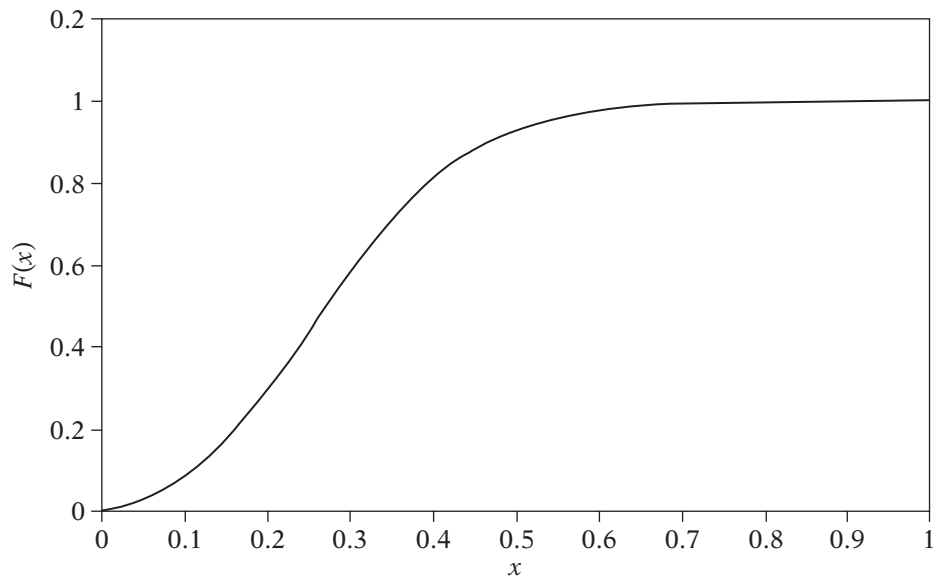


Figure 11.4 Corresponding cumulative distribution function for the probability density function in Figure 11.3.

11.2.3 Mean and Standard Deviation

Suppose we conduct N random trials with the random variable X, giving us results (or *samples*) $X_0, X_1, \ldots, X_{N-1}$. If we take what is commonly known as the average of the values, we get the *sample mean*

$$\bar{X} = \frac{1}{N} \sum_{i=0}^{N-1} X_i$$

We can think of this as representing the center of the values produced. We can get some sense of spread of the values from the center by computing the *sample variance* s^2 as

$$s^2 = \frac{1}{N-1} \sum_{i=0}^{N-1} (X_i - \bar{X})^2$$

The larger the sample variance, the more the values spread out from the mean. The smaller the variance, the closer they cluster to the mean. The square root s of this is known as the *standard deviation of the sample*.

Note that these values are computed for the samples we record. We can compute similar values for the mass or density function for X as well, dropping the reference to "sample" in the definitions.

The *expected value* or *mean* of a discrete random variable X with sample space Ω of size n and mass function $m(x)$ is

$$E(X) = \sum_{i=0}^{n-1} x_i m(x_i)$$

And for a continuous random variable, it is

$$E(X) = \int_{-\infty}^{\infty} x f(x) dx$$

Both are often represented as μ for short. Similar to the sample mean, these represent the center of the probability mass or density function, respectively.

The corresponding spread from the mean is the variance, which is computed in the discrete case as

$$\sigma^2 = \sum_{i=0}^{n-1} (x_i - \mu)^2 m(x)$$

and in the continuous case as

$$\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f(x) dx$$

As before, the square root of the variance, or σ, is called the *standard deviation*.

We'll be making use of these quantities below, when we discuss the normal distribution and the Central Limit Theorem.

11.2.4 Special Probability Distributions

There are a few specific probability mass functions and probability density functions that are good to be aware of. The first is the uniform distribution. A uniform probability mass function for n discrete random variables has $m(x_i) = 1/n$ for all x_i. Similarly, a uniform probability density function over the interval $[a, b]$ has $f(x) = 1/(b - a)$ for all $a \leq x \leq b$ and $f(x) = 0$ everywhere else. Examples of uniform probability distributions are rolling a fair die or drawing a card. On the other hand, the distribution of a loaded die is nonuniform. Similarly, our PDF in Figure 11.3 has a nonuniform distribution. Our immediate goal in building a random number generator is simulating a uniformly distributed random variable, but as the large majority of situations we deal with will have nonuniform distributions, simulating those also will be important.

There are two other distributions that are of general interest. The first is a discrete distribution known as the *binomial distribution*. Suppose we have a random experiment where there are only two possible outcomes: success or failure. How we measure success depends on the experiment: It could be rolling a 2 or 3 on a single die roll, or flipping a coin so it lands heads, or picking out a red ball. Each time we perform the experiment it must not affect any other time (i.e., it is independent), and the probabilities must remain the same each time. Now we repeat this experiment n times, and ask the question, how many successes will we have? This is another random variable, called the *binomial random variable*.

In general, we're more interested in the probability that we will have k successes, which is

$$Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

where

$$\binom{n}{k} = \frac{n!}{k!(n - k)!}$$

This is known as the *binomial coefficient*. If we graph the result for $n = 8$, $p = 2/3$, and all values of k from 1 to n, we get a lopsided pyramid shape (Figure 11.5). Note that the mean lies near the peak of the pyramid. It will only lie at the peak if the result is symmetric, which only happens if the probability $p = 1/2$.

This discrete distribution can lead to a continuous density function. Suppose that n gets larger and larger. As n approaches ∞, the discrete distribution will start to approximate a continuous density function; oddly, this function also becomes symmetric. Now we take this continuous function and translate it so that the mean lies on 0, and scale it so that the standard deviation is 1, while maintaining an area of 1 under the curve. What we end up with is seen in Figure 11.6: the *standard normal distribution*. This can be represented by the function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

We can also have a general normal distribution where we can specify mean and standard deviation, also known as a *Gaussian distribution* or a *bell curve*:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

Note that the Gaussian distribution is also the same one used (albeit in 2D) when applying a blur filter to an image or to generate a mipmap.

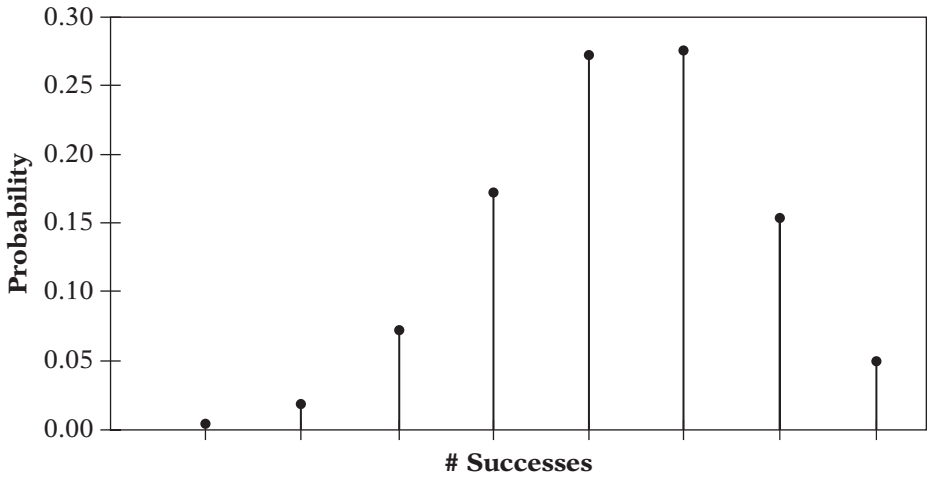


FIGURE 11.5 Binomial distribution for $n = 8$ and $p = 2/3$.

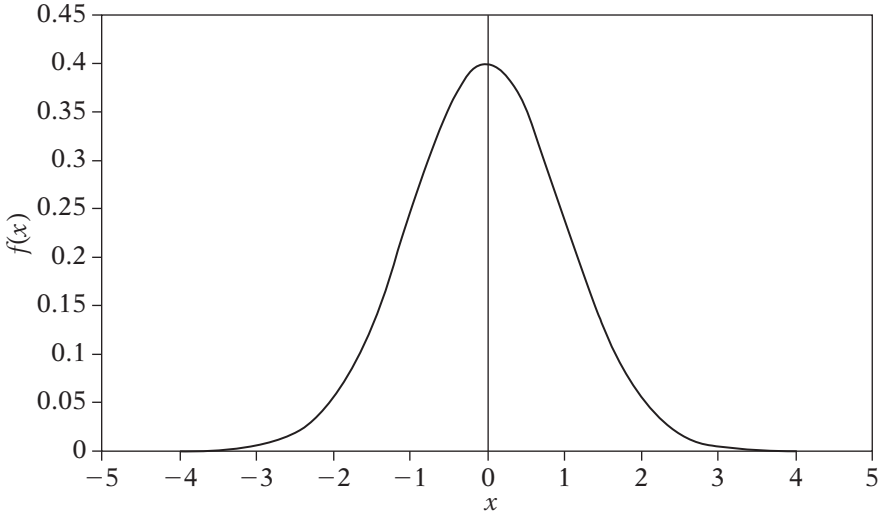


FIGURE 11.6 The standard normal distribution.

Figure 11.7 shows a general normal distribution with a mean of 3.75 and a standard deviation of 2.4. For any value of p, the binomial distribution of n trials can be approximated by a normal distribution with $\mu = np$ and $\sigma = np(1-p)$. Also, for a further intuitive sense of standard deviation it's helpful to note that in the normal distribution 68 percent of results are within 1 standard deviation around the mean, and 95 percent are within 1.96 standard deviations.

The interesting thing about the normal distribution is that it can be applied to all sorts of natural phenomena. Test values for a large group of students will fall in a normal distribution. Or measurements taken by a large group, say length or temperature, will also fall in a normal distribution.

With the introduction of the normal distribution we can also draw a better relationship between the mean and the sample mean. Suppose we take N random samples using a probability distribution with mean μ and standard deviation σ. Due to a theorem known as the Central Limit Theorem, it can be shown that the sample mean $\bar{X}$ of our samples should be normally distributed around the mean μ, and that the standard deviation of $\bar{X}$ is $\sigma/\sqrt{N}$. So, the average of random samples from a normal distribution is also normally distributed, and the larger N is, the smaller $\sigma/\sqrt{N}$ will be. So, what this is saying is that for very large N, the mean μ and the sample mean $\bar{X}$ should be nearly equal. We'll be making use of this when we discuss hypothesis testing in the next section.

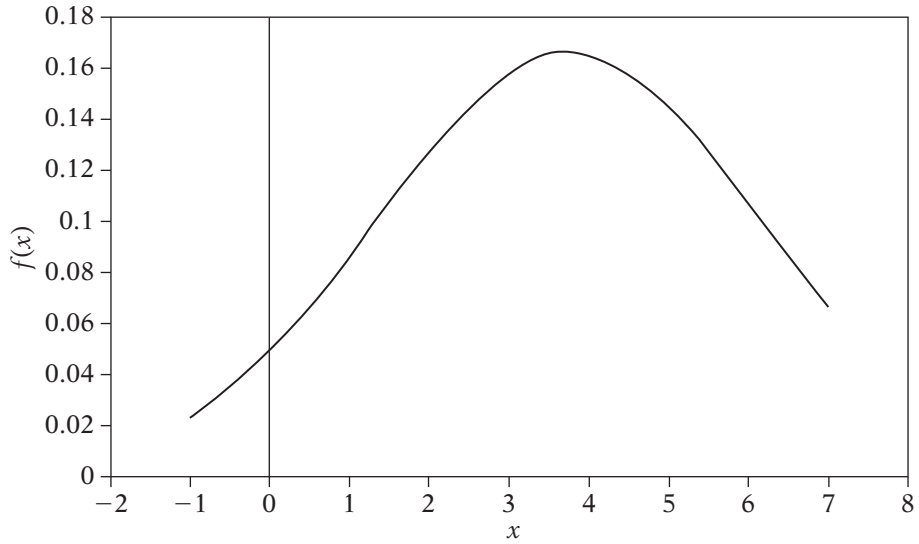


Figure 11.7 General normal distribution with mean of 3.75 and standard deviation of 2.4.

11.3 Determining Randomness

Up to this point we have been talking about random variables and probabilities while dancing around the primary topic of this chapter — randomness. What does it mean for a variable to be random? How can we determine that our method for generating a random variable is, in fact, random? Unfortunately, as we'll see, there is no definitive test for randomness, but we can get a general sense of what randomness is.

We use the term *random* loosely to convey a sense of nondeterminism and unpredictability. Note that human beings are notoriously bad at generating random numbers. Ask a large group of people for a number between 1 and 10, and the majority of the people will pick 7. The reason is that they are consciously trying to be random — trying to avoid creating a pattern, as it were — and by doing so they create a new pattern. The same can happen if you ask someone to generate a random sequence of numbers. They will tend to mix things up, placing large numbers after small ones, and avoiding "patterns," such as having the same number twice in a row. The problem is that a true random process will generate such results — streaks happen. So again, by trying to avoid patterns, a new and more subtle pattern is generated.

This gives us a clue as to how we might define a sequence of random numbers: a sequence with no discernable pattern. Statistically, when we say a

process is random, we mean that it lacks *bias* and *correlation*. A biased process will tend toward a single value or set of values, such as rolling a loaded die. Informally, correlation implies that values within the sequence are related to each other by a pattern, usually some form of linear equation. As we will see, when generating random numbers on a computer we can't completely remove correlation, but we can minimize it enough so that it doesn't affect any random process we're trying to simulate.

11.3.1 Chi-Square Test

In order to test for bias and somewhat for correlation, we will perform a series of random experiments with known probabilities and compare the results of the experiments with their expected distribution. For this comparison, we'll use a common statistical technique known as *hypothesis testing*. The way we'll use it is to take a set of observed values generated by some sort of random process (we hope), compare against an expected distribution of values, and determine the probability that the result is suitably random. Most of the tests we'll see below pick a particularly nasty test case and then use hypothesis testing to measure how well a random number generator does with that case.

The first step of hypothesis testing is to declare a *null hypothesis*, which in this case, is that the random number generator is a good one and our samples approximate the probability distribution for our particular experiment. Our *alternate hypothesis* is that the results are not due to chance — that something else is biasing the experiment.

The second step is to declare a test statistic against which we'll measure our results. In our case, the test statistic will be the particular probability distribution for our experiment.

The third step is to compute a p-value comparing our test statistic to our samples. This is another random variable that measures the probability that our observed results match the expected results. The lower this probability, the more likely that the null hypothesis is not true for our results. Finally, we compare this p-value to a fixed significance level α. If the p-value is less than or equal to α, then we agree that the null hypothesis is highly unlikely and we accept the alternate hypothesis.

One possibility for our p-value is to compare the sample mean for our results with the mean for our probability distribution. From the Central Limit Theorem, we know that the sample mean is normally distributed, and the probability of the sample mean lying outside of 1.96 standard deviations from the mean is around 5 percent. So, one choice is to let the p-value be the probability of our deviation from the sample mean, and our significance level 5 percent (i.e., if we lie outside two standard deviations we fail the null hypothesis).

However, in our case we're going to use a different technique known as *Pearson's chi-square test*, or more generally the chi-square (or χ^2) test. Chi-square in this case indicates a certain probability distribution, so there can be other chi-square tests, which we won't be concerned with in this text.

To see how the chi-square test works, let's work through an example. Suppose we want to simulate the roll of two dice, summed together. The probabilities of each value are as follows.

Die Value	2	3	4	5	6	7	8	9	10	11	12
Probability	1/36	1/18	1/12	1/9	5/36	1/6	5/36	1/9	1/12	1/18	1/36

So, if we were to perform, say, 360 rolls of the dice, we'd expect that the dice would come up the following number of times.

Die Value	2	3	4	5	6	7	8	9	10	11	12
Frequency	10	20	30	40	50	60	50	40	30	20	10

These are the *theoretical frequencies* for our sample trial. Our null hypothesis is that our random number generator will simulate this distribution. The alternate hypothesis is that there is some bias in our random number generator. Our test statistic is, as we'd expect, this particular distribution. In addition, note that we need a large number of samples in order for our chi-square test to be valid.

Now take a look at some counts generated from two different random number generators.

Die Value	2	3	4	5	6	7	8	9	10	11	12
Experiment 1	9	21	29	43	52	59	47	38	31	19	12
Experiment 2	17	24	28	29	35	76	46	35	32	23	15

First of all, note that neither matches the theoretical frequencies exactly. This is actually what we want. If one set matched exactly, it would not be very random, and its behavior would be very predicable. On the other hand, we don't want our random number generator to favor one number too much over the others. That may indicate that our dice are loaded, which is also not very random.

The first step in determining our p-value is computing the chi-square value. What we want to end up with is a value that straddles the two extremes — neither too high nor too low. Computing it is very simple: For each entry, we just subtract the theoretical value from e_i, the observed value

o_i, square the result, and divide by the theoretical value. Sum all these up and you have the chi-square value. In equation form, this is

$$V = \sum_{i=0}^{n} \frac{(e_i - o_i)^2}{e_i}$$

Using this, we can now compute the chi-square values for our two trials. For the first we get 1.269, and for the second we get 21.65.

Now that we have a chi-square value, we can now compute a *p*-value. To do that, we compare our result against the chi-square distribution. Or more accurately, we compare against the cumulative distribution function of a particular chi-square distribution.

To understand the chi-square distribution, suppose we have a random process that generates values with a standard normal distribution (i.e., a mean of 0 and a standard deviation of 1). Now let's take k random values and compute the following function:

$$\chi^2 = \sum_{i=1}^{k} {x_i}^2$$

The chi-square distribution indicates how the results from this function will be distributed. Figure 11.8 shows the probability density function and cumulative density function for various values of k.

In order to know which chi-square distribution to use, we need to know the degrees of freedom k in our experiment. This is equal to the number of possible outcomes, minus 1. In our example above, the k value is $11 - 1 = 10$. If we now substitute our computed chi-square value into the appropriate chi-square cumulative density function, that gives us the probability that we will get this chi-square value or less. This is the *p*-value we're looking for. If the resulting *p*-value is very low, say from 0 to 0.1, then our numbers aren't very random, because they're too close to the theoretical results. If the *p*-value lies in the higher probability range, say from 0.9 to 1.0, then we know that our numbers aren't random because one or more values are being emphasized over the others. What we want is a *p*-value that lies in the sweet spot of the middle. This is a slightly different approach to hypothesis testing, because we're trying to check two conditions here instead of one.

So, how do we calculate the *p*-value? This can be calculated directly, but the process is fairly complex. Fortunately, tables of pregenerated values are available (e.g., Table 11.1), and looking up the closest value in a table is good enough for our purposes.

For the particular row that corresponds to our number of degrees of freedom, we find the entry closest to our value V. The column for that entry gives us the *p*-value. Looking at the $k = 10$ column, we see that the chi-square

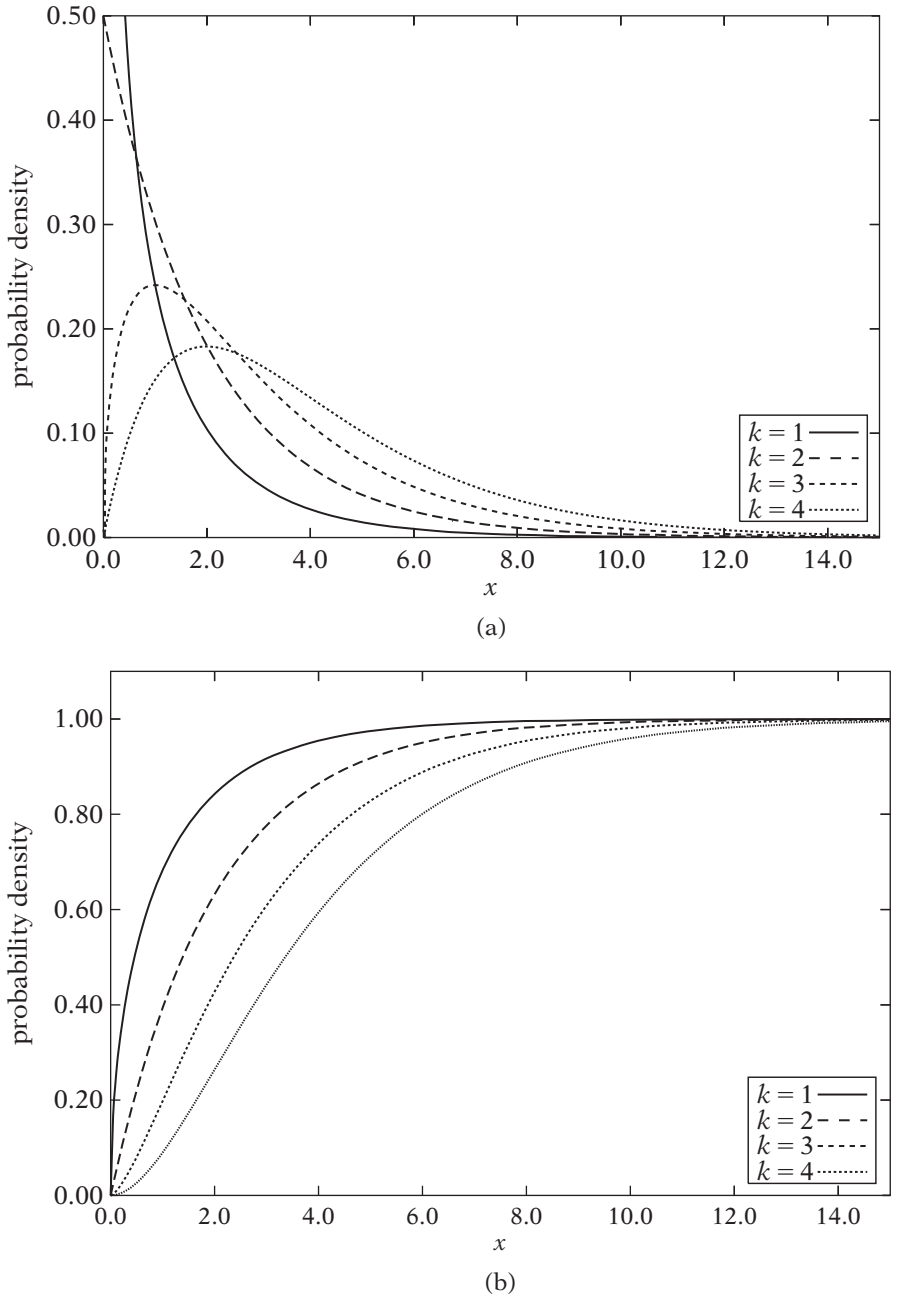


FIGURE 11.8 (a) The chi-square probability density function for values of k from 1 to 4, and (b) the chi-square cumulative density function for values of k from 1 to 4.

Table 11.1 Chi-square CDF values for various degrees of freedom k

	$p = 0.01$	$p = 0.05$	$p = 0.1$	$p = 0.9$	$p = 0.95$	$p = 0.99$
$k = 1$	0.00016	0.00393	0.01579	2.70554	3.84146	6.63489
$k = 2$	0.02010	0.10259	0.21072	4.60518	5.99148	9.21035
$k = 3$	0.1148	0.35184	0.58438	6.25139	7.81472	11.3449
$k = 4$	0.29710	0.71072	1.06362	7.77943	9.48772	13.2767
$k = 5$	0.55430	1.14548	1.61031	9.23635	11.0704	15.0863
$k = 6$	0.8720	1.63538	2.20413	10.6446	12.5916	16.811
$k = 7$	1.23903	2.16734	2.83311	12.0170	14.0671	18.4753
$k = 8$	1.6465	2.73263	3.48954	13.3616	15.5073	20.0901
$k = 9$	2.08789	3.3251	4.16816	14.6837	16.9190	21.6660
$k = 10$	2.55820	3.94030	4.86518	15.9871	18.3070	23.2092
$k = 11$	3.0534	4.57480	5.57779	17.275	19.6751	24.7250
$k = 12$	3.57055	5.22602	6.30380	18.5493	21.0260	26.2170
$k = 13$	4.10690	5.8919	7.04150	19.8119	22.3620	27.6881
$k = 14$	4.66041	6.5706	7.78954	21.064	23.6848	29.1411
$k = 15$	5.22936	7.26093	8.54675	22.3071	24.9958	30.5780

value of 1.269 for experiment 1 produces a p-value of at most 0.01, and the chi-square value of 21.65 for experiment 2 produces a value between 0.95 and 0.99. So experiment 1 is too close to the expected probability distribution, and experiment 2 is far away. This fits the way they were generated. The first set of random numbers we simply chose to be very close to the expected value. The second set were weighted so that 1 would be more likely to come up on one die and 6 more likely on the other.

An alternative to looking up the result in a table is to use a statistical package to compute this value for us. Microsoft Excel has a surprising amount of statistical calculations available, and the chi-square test is one of those. A quick online search for "chi-square calculator" also finds a number of Web applications that perform this operation. Note that Excel and most tables reverse the sense of the p-value; that is, rather than compute the probability that the chi-square value is less than or equal to our computed value, they compute the probability it will exceed that value. This allows them to use the standard approach to using p-values, where a low p-value means that our experiment is biased. Therefore, when using these packages keep this in mind.

This procedure gives us the basic core of what we need to test our random number generators: We create a test with random elements and then determine the theoretical frequencies for our test. We then perform a set of random trials using our random number generator and compare our results

to the theoretical ones using the chi-square test. If the p-value generated is acceptable, we move on, otherwise, the random number generator has failed. Note that if a generator passes the test, it only means that the random number generator produces good results for that statistic. If the statistic is one we might use in our game, that might be good enough. If it fails, it may require more testing, since we might have gotten bad results for that one run. With this in place, we can now talk about a few of the most basic tests.

The most basic test we can perform is the *equidistribution test*, which determines whether our presumably uniform random number generator produces a uniform sequence. Our test statistic is that the counts will be the same for all groups. Ideally, we set one bucket for each possible value, but given that we can have thousands of values, that's not often practical. Usually, values are grouped into sequential groups; that is, we might shift a 32-bit random number right by 24 and count values in 256 possible groups.

The *serial test* follows onto the equidistribution test by considering sequences of random numbers. In this case, we generate pairs of numbers (e.g., $(x_0, x_1), (x_2, x_3), \ldots,$) and count how many times each pair appears. Our test statistic is that we expect the count for each particular pair to be uniformly distributed. The same is true for triples, quadruples, and so on up, although managing any size larger than quadruples gets unwieldly and so something like the poker hand test, below, is recommended.

The *poker hand test* consists of building hands of cards, ignoring suits, and counting the number of poker hands, which Knuth represents as follows:

All different	*abcde*
Pair	*aabcd*
Two pair	*aabbc*
Three of a kind	*aaabc*
Full house	*aaabb*
Four of a kind	*aaaab*
Five of a kind	*aaaaa*

Each of these outcomes have different probabilities. We generate numbers between, say, 2 and 13, and track the number of poker hands of each type. Then, as before, we compare the results with the expected probabilities by performing the chi-square test.

There is a simplification of this, where we only count the number of different values in the poker hand. This becomes, then:

5 values	All different
4 values	One pair
3 values	Two pair, three of a kind
2 values	Full house, four of a kind
1 value	Five of a kind

This is both easier to count, and the probabilities easier to compute. In general, if we're generating numbers from 0 to $d-1$, with a poker hand of size k, Knuth gives the probability of r different values as

$$p_r = \frac{d(d-1)\ldots(d-r+1)}{d^k} \left\{ \begin{array}{c} k \\ r \end{array} \right\}$$

where

$$\left\{ \begin{array}{c} k \\ r \end{array} \right\} = \frac{1}{r!} \sum_{j=0}^{r} (-1)^{r-j} \left(\begin{array}{c} r \\ j \end{array} \right) j^k$$

This last term is known as a *Stirling number of the second kind*, and counts the number of ways to partition k elements into r subsets.

These three are just a few of the possibilities. There are other tests, many with colorful names such as the birthday spacing test or the monkey test. For those who want to create their own random number generators and need to run them through a series of tests, a couple of open-source libraries are available. The first is DIEHARD, created by George Marsaglia, and so named because a non-English speaker misunderstood the notion of a "battery" of tests. However, the name is appropriate, as the tests are very thorough. DIEHARD is no longer maintained, but is available online. For a regularly updated library, there is DieHarder, which was created by Robert G. Brown of Duke University. In addition to regular maintenance, this one adds some additional tests suggested by the National Institute for Standards and Technology, and is released under the Gnu Public License. It is also available online and installable on Linux as a package.

In general, however, we will not be creating our own random number generator. In those cases, a chi-square test is more useful for verifying that your use of a random number generator matches your expected behavior. For example, suppose you were trying to generate a particular probability distribution that a designer has created. If your results in-game don't match this distribution, you know you've done something wrong. The chi-square test allows you to verify this.

11.3.2 Spectral Test

There is one test of random number generators that falls outside of the standard chi-square–based tests, and that is the *spectral test*. The spectral test is derived from the fact that researchers noticed that if they constructed points in space using certain RNGs, those points would align along a fixed number of planes (a statistician would say that the data are linearly correlated).

This means that no point could be generated in the space between these planes — not very random. For many bad RNGs, this can be seen by doing a two-dimensional (2D) plot, for others, a three-dimensional (3D) plot is necessary. Some extreme examples can be seen in Figure 11.9.

In fact, Marsaglia [71] showed that for certain classes of RNGs (the linear congruential generators, which we'll cover below) this alignment is impossible to avoid. For a given dimension k, the results will lie "mainly in the planes," to quote the title of the article.

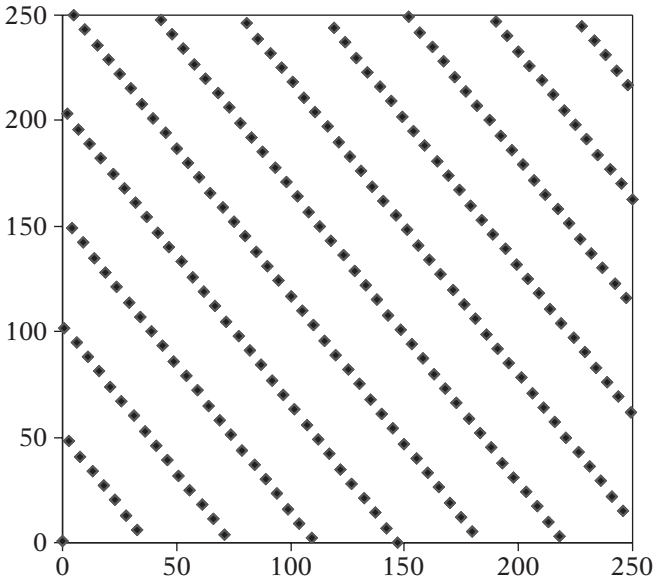
The spectral test was created to test for these cases. It takes d-tuples $(x_i, x_{i+1}, \ldots, x_{i+d-1})$ of a random sequence and looks for the spacings between them that lie along a d-dimensional hyperplane. For our purposes, we are not going to implement the spectral test. It mostly applies to a single class of RNGs, and as we'll see, a great deal of research has been done on determining good RNGs, so it's unlikely that we'll need a spectral test. Also, if the spacing between the planes is small enough, it's unlikely that it will significantly affect the sort of random data that are generated for games. However, this property of some RNGs is something to be aware of.

11.4 Random Number Generators

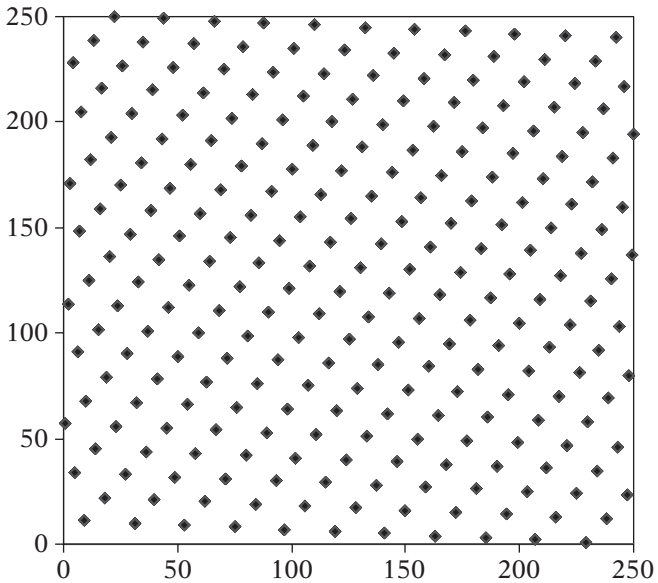
Now that we've covered some basic probability and some means of testing randomness, we can talk about how we generate random numbers. True random generators for computers are only possible by creating circuitry that depends on some physical phenomenon. One example is a generator that took video of lava lamps and used that to generate random numbers over time. Alternatively, we could track the particles generated by a radioactive isotope. Usually, however, a circuit is built that takes advantage of the fact that power to the computer has a certain amount of unpredictable noise in it. This noise is amplified and used to generate random values.

In our case, we can't assume access to such hardware. Instead, we'll have to make use of what is called a pseudo-random number generator. We will start with a set of one or more numbers and use a deterministic algorithm to generate a sequence of numbers that appear random. That is, our process is completely predetermined, but the numbers generated fulfill certain characteristics that make them suitable for simulating actual random processes. Because of this, pseudo-random number generators are just referred to as *random number generators*.

There is another class of RNGs known as quasi-random number generators. These generate numbers in a way that avoids streaks and clumping, and are primarily used for a numerical integration technique known as Monte Carlo integration. However, we won't be considering those as they tend to be more expensive and we don't require that kind of precision.



(a)



(b)

FIGURE 11.9 Examples of randomly generating points that "stay mainly within the planes."

Why study random number algorithms when most languages these days come with a built-in RNG? The reason is that these built-in RNGs are usually not very random. Understanding why they are flawed is important if we intend on using them and working around their flaws, and understanding what makes a good generator is important if we want to create our own.

Our goal in building an RNG is to generate a series or *stream* of numbers with properties close to those of actual random events. Because this series of numbers is usually very large, all of the RNGs that we're going to discuss can be described by a special type of function known as a *recurrence relation*. Those with experience in recursion should be familiar with the concept: The value at a given step n is dependent on values from previous steps (in many cases, only the immediately previous step). For example, here is the recurrence relation for the Fibonacci series:

$$x_n = x_{n-1} + x_{n-2}$$

To start things off, one or more *seed* values are set, and these control how the sequence of numbers will proceed. Again, using our Fibonacci example, using seed values $x_0 = 0$ and $x_1 = 1$, we get the series

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

The process alone doesn't produce our sequence — the seed also plays a part. For example, if we use seed values $x_0 = 2$ and $x_1 = 1$, we get the Lucas numbers:

$$2, 1, 3, 4, 7, 11, 18, 29, \ldots$$

So, choosing the proper seed value is very important. If we use the same seed all the time, we'll always get the same sequence every time. This can be useful for debugging, so that we get the same results during each debugging pass, but in the final game we'll probably want to randomize this seed value somehow. One common method is to use the operating system clock value. Another uses the frequency of the user's keystrokes, mouse movement, or joystick movement at start-up time to compute a random value for the rest of the game.

The Fibonacci series is infinite, since the values get progressively larger and larger. However, we will need to limit our results to fit within calculable values on the computer, so we will take a modulus of anything we compute to ensure that it stays within bounds. Doing this with Fibonacci gives us

$$x_n = (x_{n-1} + x_{n-2}) \mod m$$

The value m is often one more than the largest representable number, although as we'll see below, other values work better with certain algorithms.

Another final concept we need to discuss before diving in is the *period* of a random number sequence. Because of the modulus, eventually all generators will repeat their values; you will end up generating your original seed values and the sequence will start again. For example, take this (very poor) RNG (please):

$$x_n = (x_{n-1} + 2) \mod 4$$

Given a seed value of 0, this will generate the sequence

$$0, 2, 0, 2, 0, 2, 0, 2, \ldots$$

This is a poor RNG for two reasons. First, as we can see, the values are very regular. But also, it has a very small period of 2. We want this period to be as large as possible; at the very least it should encompass all values $(0, \ldots, m-1)$, and ideally be much larger than that so that we can get streaks of numbers.

This should give some general sense of the structure of the algorithms we'll be discussing. Note that this is by no means an exhaustive list. We are merely trying to present some standard algorithms to demonstrate the wide variety of possibilities. A few of the generators we'll discuss are not very good. This is partially to show what can go wrong in case you are tempted to create your own, and partially to build up the background to understand the best current generator: the Mersenne Twister. Also note that when discussing generators in this section, we'll only be constructing those that generate unsigned integers. We'll cover how to create signed integers, smaller than full integer ranges, and floating-point numbers in a later section.

11.4.1 Linear Congruential Methods

Definition

SOURCE CODE
LIBRARY
IvRandom
FILENAME
IvLCG64
IvCGPrime

The *linear congruential generator* (LCG) is a very popular random number generator. It was first introduced by D. H. Lehmer in 1949 and is introduced in most algorithm classes and implemented in most standard libraries. The LCG is represented by the following equation:

$$x_n = (ax_{n-1} + c) \bmod m$$

where

$$0 \leq m$$
$$0 \leq a < m$$
$$0 \leq c < m$$

In this case, m is called the *modulus*, a is called the *multiplier*, and c is called the *increment*. If c is 0, this is called a *multiplicative congruential method*, otherwise it is a *mixed congruential method*.

Note that no matter what the values are, the maximum period is m. This makes sense; because we're only tracking one variable, if we ever repeat a value the sequence will begin again from that point. So, the maximum we can possibly do is to run through all of the values between 0 and $m - 1$ and then start again from the seed. Of course, this is only possible if $c \neq 0$. If $c = 0$, then if our sequence generates 0, we'll end up with something like

$$\ldots, 4, 24, 6, 0, 0, 0, 0, 0, \ldots$$

This is because once x is 0, $ax \bmod m$ will always be 0. So if $c = 0$, we can only use values between 1 and $m - 1$. In this case, if we need 0, we can always subtract 1 from the result.

Choosing the Modulus

The first question when constructing a LCG is what the value of m should be. The most obvious choice, as we stated above, is to use one more than the largest representable integer value, or, if our word size is w bits, 2^w. As most adders will wrap values when overflow occurs, they are automatically performing a modulus 2^w, which makes our computation very efficient.

However, there are two problems when using a power of 2 for m. First, it can be shown that if $c = 0$, then the maximum period is only $m/4$, and this in turn can only happen if $a \bmod 8$ is 5 and the initial seed is odd. Since we're multiplying an odd number by an odd number, the result will be an odd number, and in fact, we'll get half of the odd numbers between 1 and $2^w - 1$. To avoid this, most generators that use $m = 2^w$ use an odd value for c, which allows the resulting value to alternate between even and odd, and provides the full period of m.

However, when our numbers alternate between even and odd, the least significant bit will alternate between 0 and 1 — not very random. This signifies another problem, as pointed out by Knuth [65]. Suppose that d is a divisor of m (i.e., $m \bmod d = 0$) and

$$y_{n-1} = x_{n-1} \bmod d$$

If $d = 2^e$, we can think of this as representing the eth least significant bits of x_{n-1}. It can be shown that

$$y_n = (ay_{n-1} + c) \bmod d$$

In other words, while our random sequence may have a maximum period of m, its least significant bits have a maximum period of d — they are much less random than the most significant bits.

This really only comes into play if we're using our RNG for small value simulations such as rolling dice. One solution is to shift the result from the RNG to the right so that the more random bits in the middle word become the new least significant bits and then take our modulus, which is what we have done in ours.

However, because of these problems, most researchers recommend using a large prime number for m instead. There is some debate on what that suitable prime number is. One popular choice for 32-bit numbers is $2^{31} - 1$, because it is close to our maximum value of $2^{32} - 1$ (which is not prime, by the way). Marsaglia [72] also suggests $2^{32} - 5$ and $2^{32} - 2$. However, in general, we simply want a large enough prime for our purposes — the larger the prime, the larger the period.

The big advantage of the prime value is that it guarantees to give us a full period without having to use an increment. However, using a prime value has some consequences. First of all, we need to be sure to store the maximum possible value somewhere, so that we can compute the floating-point equivalents (see below). We also lose the convenience and efficiency of letting the hardware magically handle our modulus for us. In fact, rather than taking advantage of overflow, we have to be careful that it doesn't take advantage of us. For example, suppose we have a 4-bit architecture, and are computing the following LCG:

$$x_n = (3x_{n-1}) \bmod 13$$

If x_{n-1} is 12, then $3x_{n-1} = 36$. But this value doesn't fit into the word, so it is truncated to $36 \bmod 16 = 4$. This mod 13 gives a final value of 4. But the actual result should be $36 \bmod 13 = 10$.

To solve this problem, Park and Miller [89] recommend the following replacement formula, based on Schrage [101], for $ax \bmod m$:

$$ax \bmod m = a(x \bmod q) - r(x \div q) + m\delta(x)$$

where $\div$ is integer division, $q = m \div a$, $r = m \bmod a$, and $\delta(x) = (z \div q) - (az \div m)$. The value of $\delta(x)$ will either be 0 or 1, and only 1 if $a(x \bmod q) - r(z \div q)$ is less than 0. We can represent this in code as follows:

```
x = a*(x%q) - r(x/q);
if (x <= 0)
    x += m;
```

Note that this only works if $r < q$, otherwise overflow will still occur.

Of course, a simpler solution is to do our calculations in a larger word size and truncate down to our desired, smaller word size (i.e., compute in 64-bit integers for a 32-bit result), but that assumes this option is available to us.

Choosing the Multiplier

So these are our two logical possibilities for a modulus: either a power of 2 or a large prime number. We've already noted that $c > 0$ is only necessary when using a modulus that is a power of 2, and in that case any odd number will do (1 is a popular choice). So, the remaining question is: What do we choose for a, our multiplier?

We want to make our choice to maximize two things: the period of the random sequence and the randomness of the resulting numbers within that sequence. The most common measure of this randomness is to use the spectral test, as LCGs are particularly susceptable to obviously regular patterns unless the values are chosen appropriately.

Let's consider the maximum period first. We've already noted that if $m = 2^w$, we want $a \mod 8 = 5$ and c to be odd to get the full period. In general, however, we want the following. Suppose we begin with seed x_0, multiplier a, modulus m, and $c = 0$. Then we can find the value at the ith step by calculating

$$x_i = a^i x_0 \bmod m$$

Without loss of generality, let's assume that $x_0 = a$. If we have a full period, eventually we'll generate a as our current random number, and that starting point is as good as any other. So, this formula becomes

$$x_i = a^{i+1} \bmod m$$

In order for us to get a full cycle, $a \bmod m$, $a^2 \bmod m$, $a^3 \bmod m$, $\ldots$, $a^{m-1} \bmod m$ must all be distinct values. How do we find this a? Fortunately, there is a mathematical entity known as a *primitive element*, which has just this property when a and m are relatively prime (that is, their only common factor is 1). So, for our particular m, we just need to generate all the primitive elements, and that will give us good values for a.

This gives us a starting point for both cases. However, the number of possible values is still quite large and narrowing this down requires the use

of the spectral test. Fortunately for us, many people have already done studies of the primitive elements for specific values of m. A fairly recent work by L'Ecuyer [67] in particular has laid out tables of possible values for all of the cases we're interested in, including the power of two cases with no addition. Using a value from these tables will guarantee excellent results. For our generators, we have chosen default values of $a = 2862933555777941757$ for the 2^{64} generator and m of $2^{32} - 5$ and a of 93167 for the prime generator. While those values will produce good results, we of course let users set their own values if they want.

Summary

In summary, the LCG is the most commonly taught and commonly used RNG today. It is usually the basis for RNGs in most math libraries. Although it's not the best generator, when the values for a and m are chosen carefully, it can produce results good enough for most games. However, when using it, one needs to be wary of the limited period, the randomness of the least significant bits, and the problems with the spectral test. Because of this, we'll be looking at some other possibilities for RNGs.

11.4.2 Lagged Fibonacci Methods

The linear congruential methods are reasonable RNGs, but they do have their flaws. The most major flaw is that when performing the spectral test for k-dimensional points, the best we can do is to limit the number of distinct hyperplanes that points will fall on to $m^{1/k}$. In other words, there will always be points in space that we cannot randomly generate. The other problem is that the maximum period we can expect is m. Can we do better?

One thing we note about LCGs is that they only make use of the last value — perhaps we can do better by looking at more than one previous value. We've already mentioned the Fibonacci method, where we take values from the previous two steps. Recall that this has the recurrence relation

$$x_n = (x_{n-1} + x_{n-2}) \bmod m$$

and requires two seed values. For the traditional Fibonacci sequence, this would be $x_0 = 1$ and $x_1 = 1$.

Unfortunately, while the standard Fibonacci method has a large period, it has been shown to not produce very random numbers. There is actually a hidden pattern, where the ratio between one value and the previous value is approximately the golden ratio: $(\sqrt{5} + 1)/2$.

A better approach is to use a *lagged Fibonacci generator*, where we look further back into the sequence for values, and they are not necessarily one after the other. This can be generalized as

$$x_n = (x_{n-j} \star x_{n-k}) \bmod m$$

where $\star$ is any binary operation (addition, subtraction, multiplication, and exclusive-Or are common choices) and $0 < j < k$. Assuming that $m = 2^w$ and addition, subtraction, or exclusive-Or is used, the maximum possible period for lagged Fibonacci generators is $2^{w-1}(2^k - 1)$. For multiplication, this drops to $2^{w-3}(2^k - 1)$. However, multiplication has been shown to mix bits better. In any case, assuming k is large enough, this period can be much larger than a standard LCG.

There are two decisions that we have to make when dealing with lagged Fibonacci generators: What are the values of j and k, and how do we initialize the starting k values? As far as the first question, which will determine the actual maximum period, tables of good values can be found in Knuth [65]. The choice of our initial values can be even more critical, as choosing poorly will seriously affect the randomness of the resulting sequence (e.g., consider what would happen if all the initial values were the same). One good possibility presented by Mitchell and Moore is to use addition with $j = 24$ and $k = 55$. The values $x_0 \ldots x_n$ are initialized with arbitrary integers, but guaranteed to be noneven. This gives us a period of $2^{w-1}(2^{55} - 1)$, which is quite respectable. In general, however, the problem of choosing good starting values haunt the Fibonacci generator, so again we will look for other solutions.

11.4.3 Carry Methods

One of the flaws of the LCG is that it works best with a value m that is prime. This is bad for two reasons: It's not cheap to compute and computer word sizes are powers of two (which are definitely not prime). For those values that are a power of two, the least significant bits have a lower period than the entire sequence.

One solution was presented by George Marsaglia and Arif Zaman [74], who noted that with the LCG, the most significant bits get mixed better than the least significant bits. To mix the least significant bits, they proposed a carry or borrow operation, which takes part of the result from the previous stage and carries it forward to be applied to the least significant bits in the next stage. The standard formula for an add-with-carry generator is

$$x_n = (x_{n-k} + x_{n-r} + c_{n-1}) \bmod m$$
$$c_n = (x_{n-k} + x_{n-r} + c_{n-1}) \div m$$

Again, $\div$ represents an integer divide. As we can see, the bits that would normally be cast out from the modulus operation are added to the next stage, thereby mixing the lower bits. Something similar can be done with the subtract-with-borrow generator:

$$x_n = (x_{n-k} - x_{n-r} - c_{n-1}) \bmod m$$
$$c_n = (x_{n-k} - x_{n-r} - c_{n-1}) \div m$$

While these generators have large periods, solve the least significant bit issue, and otherwise show some promise, it was shown that they also fall prey to the same problem of falling mainly in the planes that linear congruent generators do.

In 1992, George Marsaglia posted a modification of this technique [73], which he called the Mother of All Random Number Generators, also known as the multiply-with-carry technique. This algorithm also works much better with values of m that are powers of 2. As before, the idea is to add the high-entropy bits (i.e., those that are changing a lot) to the low-entropy bits (i.e., those that don't normally change all that much). If we represent a 32-bit integer x_i as two 16-bit integers $a_i b_i$ (i.e., a is the high bits and b is the low bits), then a basic example of his algorithm is

$$x_n = 30903 \cdot b_{n-1} + a_{n-1}$$

Instead of returning x_n as the result, we return the low bits b_n as a 16-bit integer.

This can easily be represented in C as follows.

```
k=30903*(k&65535)+(k>>16);
return(k&65535);
```

In this case we're doing a carry mod 2^{16}.

This has a period of 2^{59} 16-bit numbers. We can generate 32-bit numbers with the same period by concatenating two results together.

```
k=30903*(k&65535)+(k>>16);
j=18000*(j&65535)+(j>>16);
return ((k << 16) + j);
```

The multipliers for j and k are chosen to give good results for a modulus of 2^{16}.

This can be extended further, giving a period of 2^{118}.

```
k=30903*(k&65535)+(k>>16);
j=18000*(j&65535)+(j>>16);
i=29013*(i&65535)+(i>>16);
l=30345*(l&65535)+(l>>16);
m=30903*(m&65535)+(m>>16);
n=31083*(n&65535)+(n>>16);
return((k+i+m)<<16)+j+l+n);
```

This is a considerable improvement over the previous two methods: It gives us very large periods, it does a good job of randomizing the bits, it works well with computer word sizes and so is good for both floating-point numbers and integers, and it's very fast. It also only requires six starting values, as opposed to the large table needed for the lagged Fibonacci methods.

An alternative 32-bit implementation by Agner Fog [37] only uses four starting values and has a period of 3×10^{47}.

```
UInt64 S = 2111111111*X[n-4] + 1492*X[n-3]
           + 1776*X[n-2] + 5115*X[n-1] + C;
X[n] = (UInt32)(S);
C = S>>32;
```

The initial values of `X[]` and `C` are randomized. Note that this assumes that we have access to a 64-bit integer, or at the very least the carry bits from the accumulator after calculating `S`. `C` in this case is the upper carry bits of the 64-bit integer, and we truncate the result to get our random 32-bit integer `X[n]`. This is very efficient and produces very good results with very little space, but as we will see, if we're willing to give up a little more memory we can do better still.

11.4.4 Mersenne Twister

Source Code
Library
lvRandom
Filename
lvMersenne

Up to this point in our discussion, the improvements in RNGs have been incremental: the period improves by a couple of orders of magnitude, the randomness by a bit more. For most interactive purposes any of these RNGs could do. However, as Bill Cosby once said, "I told you that story so I could tell you this one." In 1997, a huge leap was made in random number generators: the Mersenne Twister.

The Mersenne Twister is our holy grail—an RNG with full coverage of integers that produces random numbers good enough for anything other than

cryptological uses and that is efficient. Its only negative is that it requires a buffer of 624 integers, but for a game, this is a drop in the bucket compared to the quality that we receive.

The Mersenne Twister was developed by Matsumoto and Nishimura in 1997, building on Matsumoto and Kurita's work with generalized feedback shift register (GFSR) algorithms [75]. These are a subclass of lagged Fibonacci algorithms that use exclusive-or (represented as $\oplus$) as their binary operation. They begin with a table of n values. To generate new values, the following recurrence is used:

$$x_{l+n} = x_{l+m} \oplus x_l$$

where $l = 0, 1, 2, \ldots$. This is usually implemented by overwriting the original table as follows.

```
void initializeRandom()
{
    l = 0;
    for (int i = 0; i < n; ++i)
    {
        x[i] = rand(); // or some suitable value
    }
}
int myRand()
{
    int retValue = x[l];
    x[l] = x[(l+m) mod n] ^ x[l];
    l = (l+1) mod n;

    return retValue;
}
```

This method is very fast, but it has a few problems. First of all, how we create the table has a direct effect on how the algorithm performs. For example, suppose we initialized it with the integers from 1 to n, or even worse, all with a single value. A few moments thought will show that this will not produce very random values. Secondly, it can be shown that the period of this sequence is far smaller than the theoretical upper bound $2^{nw} - 1$. To achieve this upper bound we'd have to use a much larger table.

Matsumoto and Kurita's insight was to realize that randomness and the period can be improved if you transform or "twist" some of the bits before performing the exclusive-or. This can be represented as.

$$x_{l+n} = x_{l+m} \oplus x_l\mathbf{A}$$

where A is called the *twist matrix*. This is equal to

$$\mathbf{A} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \ddots & \\ a_0 & a_1 & \cdots & a_{n-1} \end{bmatrix}$$

This in turn boils down to a sequence of simple bit operations:

```
A(x) = (x >> 1) ^ ((x & 0x01) ? a : 0);
```

where a is a special constant value.

The performance of the twisted GFSR (TGFSR) algorithm is quite good, at the same level as the simple multiply-with-carry algorithm described above.

The one problem with this algorithm is that it still suffers from one problem that we saw with the linear congruential algorithm: While whole words may have pseudo-random properties, bit sequences within those words do not. Again, as an example, the lower r bits of the random sequence might have a period much less than $2^{nw} - 1$. To solve this, Matsumoto and Kurita added *tempering* to the basic algorithm. Instead of outputting the raw data from the table, they apply some shift and binary "and" operations to ensure that the subword sequences are also well randomized. Their suggestion is as follows.

```
y = (x[1] ^ ((x[1]<<s)& b);
y = (y ^ ((y<<t)& c);
```

In this case, y becomes our new return value, s and t are shift value constants, and b and c are new word-sized constants.

Matsumoto, with his new partner Nishimura, didn't stop there. Recall that the multiply-with-carry algorithm gets its power from mixing the bits between subsequent values. The improvement to TGFSR is to do something similar, using the following formula:

$$x_{k+n} = x_{k+m} \oplus (x_k^u | x_{k+1}^l) A$$

where $k = (0, 1, \ldots)$. As we can see this is very similar to the TGFSR algorithm, with the addition of some odd notation. The operator | represents standard "or." There is also a value r that controls how the bits are mixed. Assuming that w is the word size for x, then x_k^u is the upper $w - r$ bits of x_k, and x_{k+1}^l is the

lower r bits of x_{k+1}. So, this operation takes the upper bits of one entry and the lower bits of the following entry and concatenates them to form a new entry, just like the multiply-carry operation.

To give final better results, they also do tempering like the other TFGSR improvement, with a slightly different algorithm.

```
y = x ^ (x >> u);
y = (y ^ ((y<<s)& b);
y = (y ^ ((y<<t)& c);
y = y ^ (y >> l);
```

Note that u and l in this case are different from the u and l in the recurrence relation.

This final change gives us the Mersenne Twister [76]. The standard algorithm MT19937 uses the suggested values of $w = 32$, $n = 624$, $m = 397$, $r = 31$, $u = 11$, $s = 7$, $t = 15$, $l = 18$, $a = 0x9908B0DF$, $b = 0x9D2C5680$, and $c = 0xEFC60000$. This gives us a period of $2^{19937} - 1$ (this is one of the so-called Mersenne primes, hence the name) and a 623-dimensional spectral distribution, which is far better than any of the methods we have already discussed.

There have been some additional modifications to the Mersenne Twister algorithm that make it more efficient on SIMD processors (and also faster on serial processors as a side effect). However, the basic algorithm remains the same. It's very efficient, avoids any practical issues with the spectral test, and has good bit distribution. It also passes the standard set of chi-square–based tests. Its only downside is the large table of seed values required. If memory is an issue, then one of the other methods would have to be used. Otherwise, the Mersenne Twister more than meets our requirements for an RNG.

11.4.5 Conclusions

This concludes our discussion of basic random number generators. The question remains, which one to use? Obviously, in the best possible cases, we would use MT19937. However, as mentioned, this assumes that we have the space to store the 624-integer table, and, for example, a handheld application might not have that luxury. In that case, the multiply-carry method probably would be more than good enough. And if space and speed are truly at a premium, a linear congruential method may do the trick, but be careful to choose one with a good modulus and multiplier.

So to summarize, whenever possible, use the Mersenne Twister; be sure to pick good random seeds; and if you can ever help it, don't use the default generator — create your own or use a well-vetted library.

11.5 Special Applications

Up to this point, we've been discussing only how to randomly generate uniformly distributed unsigned integers. However, randomness in a computer game extends beyond this. In this section we'll discuss a few of the more common applications and how we can use our uniform generator to construct them.

11.5.1 Integers and Ranges of Integers

In addition to unsigned integers, it is useful to be able to generate other types of values, and in various ranges. In this section we'll discuss some of the possibilities and how to generate them. For the sake of this discussion, we will assume that we are generating values from 0 to $m-1$: If 0 is not possible with our generator, we can simply subtract 1 from the result and substitute $m-2$ for $m-1$.

If our generator has $m = 2^w$, then generating signed integers is simply a case of recasting the unsigned result as signed. The alternative is to do a scale and translate transformation, so $y = 2x - m$. This assumes that $m < 2^{w-1}$, otherwise we'll end up overflowing.

Another common case is generating a range of integers, say from a to b. If x is the result of our RNG, we could do

$$y = x \bmod (b - a + 1) + a$$

The problem is, if $b-a+1$ is small and the RNG has poor mixing of the least significant bits, the result will not be very random. One solution is to shift x to the right before performing the modulus. Another, of course, is to use a different generator. But if $b-a+1$ is large, then again we need to worry about overflow.

Because of these issues, the standard solution is to calculate the result in floating point using a floating-point uniform variate u, where u lies in $[0, 1]$, and cast the result to the integer as follows:

```
y = (int) u*(b-a)+a
```

If u lies in the interval $[0, 1)$—that is, the value 1 is not included—then we use

```
y = (int) u*(b-a+1)+a
```

This does cost more than the integer-only version, but it handles the bit-mixing and overflow problems nicely.

11.5.2 Floating-Point Numbers

Usually when generating floating-point numbers, we want the range [0, 1]. Commonly, this is computed as

```
float f = float(random())*RECIP_MAX_RAND;
```

where `RECIP_MAX_RAND` is the floating-point representation of 1 over the maximum possible random number.

An alternative is to set the exponent of the floating-point number to *bias* + 1 (see Chapter 1 for the definition of floating-point bias) and take random bits from the integer to fill the mantissa. This gives a value in the interval [1, 2). Subtracting 1 gives us an interval of [0, 1). For a single-precision floating point this can be computed as

```
unsigned int floatInt = 0x3f80000 | random() >> 9;
float f = *(float*)(&floatInt) - 1.0f;
```

If the pointer cast does not compile efficiently, we can use a union to do the bit conversion.

Now that we have values from 0 to 1, computing a general random interval [*a*, *b*) is simple:

```
y = (b-a)u + a;
```

As mentioned above, we can also use this to generate intervals for integers via casting.

11.5.3 Nonuniform Distributions

Up until now, we've only been considering uniform random numbers. However, as we've seen, a large class of random events have nonuniform distributions. How then do we calculate these?

If we have a discrete random variable and its distribution, then we can create a discrete CDF and store the results in a table. If we then roll a uniform

floating-point number in the interval [0, 1), we can then find the minimum entry that is greater than that value, and generate that. Let's take our ball drawing problem as an example. Figure 11.10 shows the CDF for the probability distribution in Figure 11.2. Notice that due to the discrete nature of the distribution the CDF is a step function. If, for example, we randomly generate the value 0.43, we find that value in the *y*-axis, and then trace along horizontally until we hit a step. Sliding down to the *x*-axis, we see that step begins at 2, which represents the color blue, so that is the result of our random variable.

If we have a continuous random variable, this is not as simple. However, we can observe that what we're doing with the discrete case is just inverting the CDF. Assuming that there is an inverse, we can do the same with the continuous CDF, plug in our uniform value, and take the result as our nonuniform random variable.

If there is no inverse or we don't know the exact function for either the PDF or the CDF, then there is one other technique we can try: the rejection method or *rejection sampling*. The idea is that we generate values using a PDF that is close to our unknown one, and then throw out those that don't match. We'll see some examples of this in the next two sections. However, this may not be the most efficient method, and in games it can be better to find an approximation of our distribution and use that.

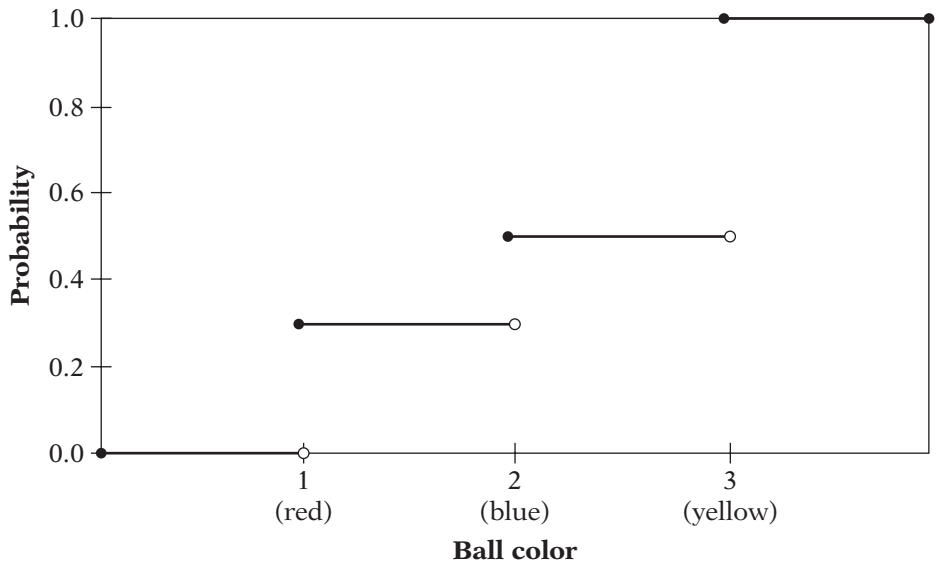


FIGURE 11.10 Discrete cumulative distribution function for the probability mass function in Figure 11.2.

11.5.4 Spherical Sampling

One common example of randomness in a game is generating the initial random direction for a particle. The most commonly used particle system of this type is spherical, where all the particles expand from a common point. We can compute the direction vector for this easily by generating a random point on a unit sphere.

One possible (but wrong) solution for this is to generate random components (v_0, v_1, v_2), where each v_i is a floating-point value in the range $[-1, 1]$, and then normalize the result. This will produce random points on the sphere but the result will not be evenly distributed across the surface of the sphere. If we look at Figure 11.11, we can see the result. Because the initial random numbers generated are within a cube, the result on the sphere is biased toward the locations closest to the corners.

Another possibility is to use rejection sampling. We again generate our three values, but then test to see if $v_0^2 + v_1^2 + v_2^2$ is less than one. It works fine,

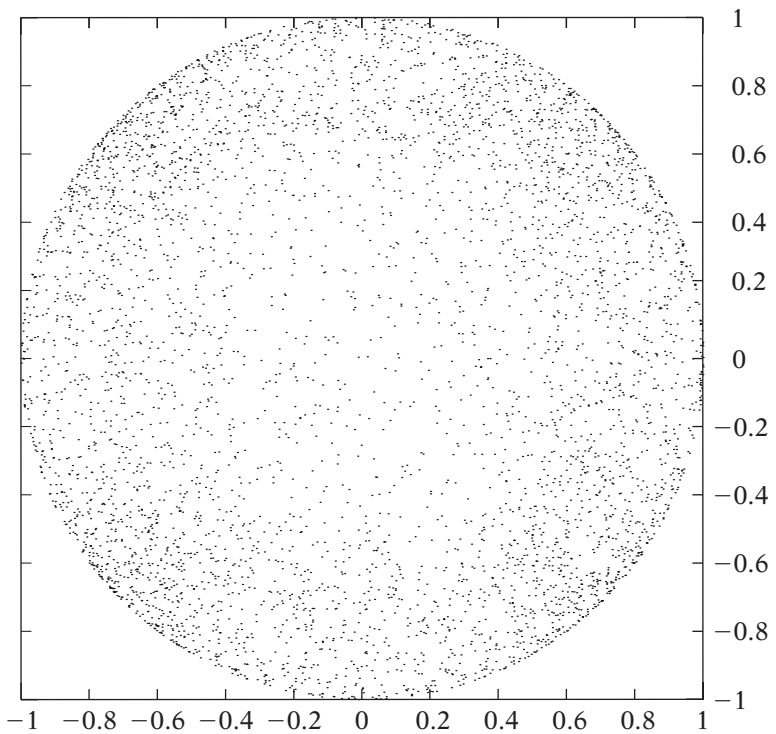


Figure 11.11 Spherical sampling, looking down along z. Result of normalizing random cube values; the points tend to collect near the "corners."

but can require a large number of RNG evaluations, so we'll consider one other option.

Rather than using Cartesian coordinates, let's look at spherical coordinates, which may be a little more natural to use on (say) a sphere. Recall that ϕ is the angle from the z-axis down, from 0 to π radians, and θ is the angle from the x-axis, from 0 to 2π radians. Since we're talking about a unit sphere, our radius ρ in this case is 1. So, we could generate two values ξ_0 and ξ_1 in the interval $[0, 1]$, and compute $\phi = \xi_0\pi$ and $\theta = 2\xi_1\pi$. From there we can compute x, y, and z as

$$x = \sin\phi\cos\theta$$
$$y = \sin\phi\sin\theta$$
$$z = \cos\phi$$

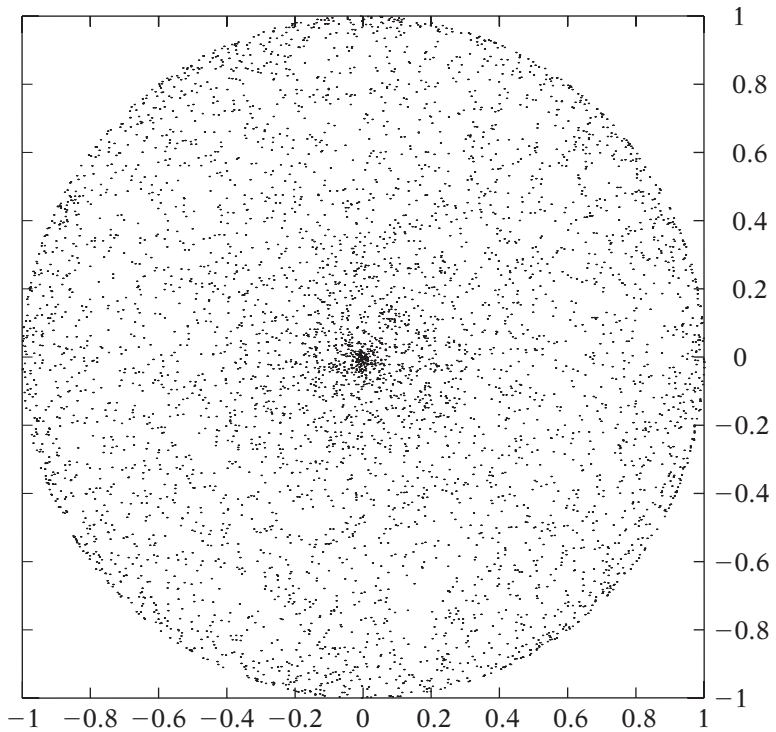


FIGURE 11.12 Spherical sampling, looking down along z. Result of randomizing spherical coordinates; the points tend to collect near the poles.

However, again we don't quite get the distribution that we expect. In Figure 11.12, we see that the points are now clustered around the poles of the sphere. The solution is to note that we want a latitude/longitude distribution, where z is our latitude and is uniformly distributed, and θ is longitude and also uniformly distributed. The radius at our latitude line will depend on z—we want to guarantee that $x^2 + y^2 + z^2 = 1$. The following calculation handles this:

$$
\begin{aligned}
z &= 1 - 2\xi \\
r &= \sqrt{1 - z^2} \\
\theta &= 2\pi\xi_1 \\
x &= r\cos\theta \\
y &= r\sin\theta
\end{aligned}
$$

The final result can be seen in Figure 11.13.

A similar calculation can be done if we want to generate points on a hemisphere. Instead of calculating $z = 1 - 2\xi$, we want z to vary from 0 to 1, which is just ξ_0.

Whether this spherical coordinate method or rejection sampling is faster depends on the cost of the square root and trigonometric functions. On one system, for example, four to five uniform variates can be generated in the time of a single trigonometric call. Profiling will be required in your particular application to determine which is best.

11.5.5 Disc Sampling

Another particle or ray casting shape that we might use is a cone. We can simulate a cone by using the cone tip as the source of our ray or particle and randomly selecting a point on the disc at the other end of the cone. This can be generalized by selecting a point on a unit disc. Afterwards we can scale the result by the radius of the cone, and then rotate it to be normal with the cone direction.

To select a point on a unit disc we could use rejection sampling again. The rejection sampling approach is similar to the 3D case: We generate two random numbers with range from 0 to 1. This time if the vector generated has length *greater* than 1 we try again, otherwise, we proceed as before.

Alternatively, we can generate a value using polar coordinates. The naive approach is to generate two values ξ_0 and ξ_1 as in the spherical case. This time we want the radius to vary from 0 to 1, and θ to vary from 0 to 2π,

and so

$$
\begin{aligned}
r &= \xi_0 \\
\theta &= 2\pi\xi_1 \\
x &= r\cos\theta \\
y &= r\sin\theta
\end{aligned}
$$

However, we find that we get clustering in the center, as we did in the spherical case (Figure 11.14). This may be close to what we want if we're calculating bullet trajectories, where we want them to cluster around the aim direction. However, let's assume this is undesirable. The insight here is to set $r = \sqrt{\xi_0}$. This pushes the values back to the edges of the disc and gives us uniform sampling across the area of the disc (Figure 11.15).

Again, profiling will be needed to determine if this method or rejection sampling will be more efficient.

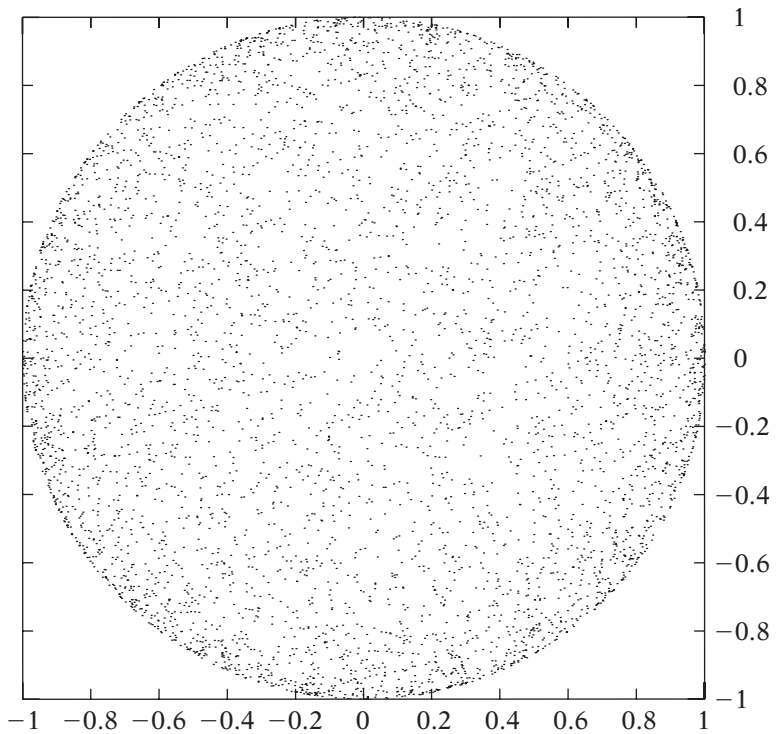


FIGURE 11.13 Spherical sampling, looking down along z. Result of randomizing latitude and longitude; the result is correct.

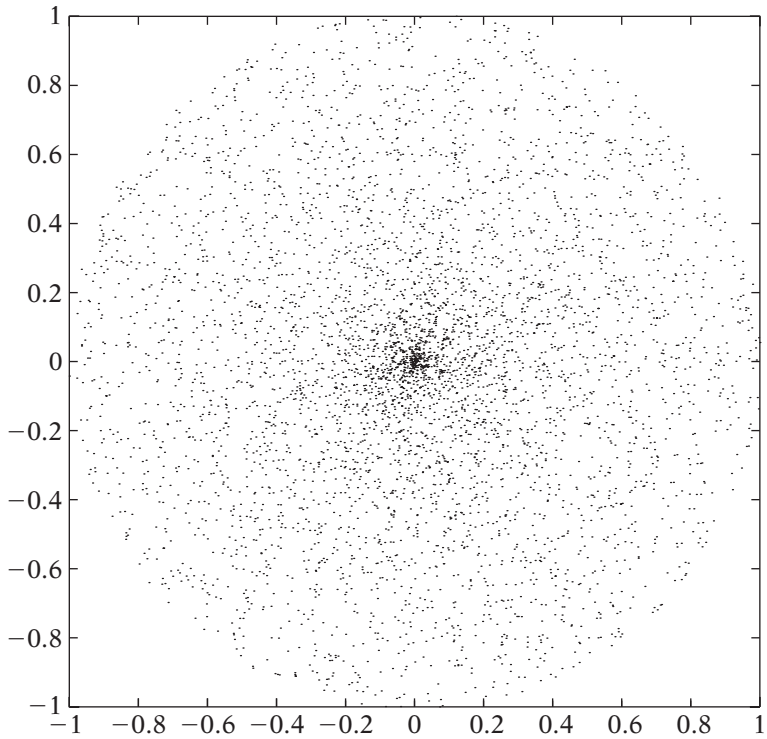


FIGURE 11.14 Disc sampling. Result of randomizing polar coordinates; the points tend to collect at the center.

11.5.6 NOISE AND TURBULENCE

SOURCE CODE DEMO Perlin

We will conclude our discussion of random numbers by briefly looking at some common noise functions and how they can be used to generate procedural textures. The first question is: Why do we want to add randomness to our procedural textures? The main reason is that the world itself is random. Random bumps against the wall create scuffmarks and divits. The way trees grow depend on rain, wind, and sun. Clouds in turn are dependent on humidity and wind. So, by adding random elements to textures that simulate natural features, we make them look less synthetic and more organic.

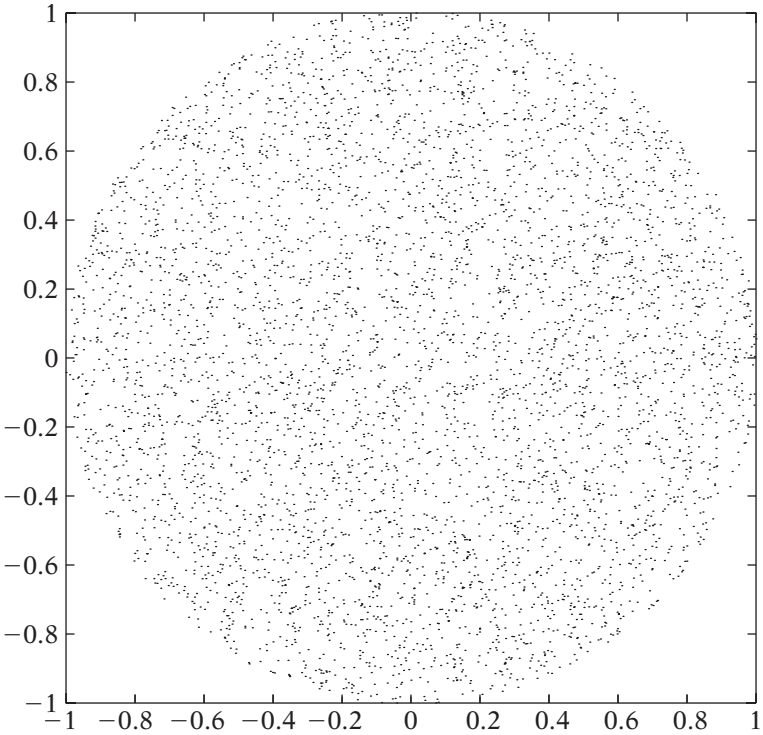


FIGURE 11.15 Disc sampling. Result of randomizing polar coordinates with radius correction; the result is correct.

The common way to apply noise to textures is to build a noise lattice. In this case, we place random values at regular intervals in the texture space, and then interpolate between them to obtain the intermediary values. By using an appropriate interpolation function (usually cubic), we can guarantee that our noise function is continuous and smooth, which produces much better visual results. This lattice can be 1D, 2D, 3D, or even 4D. Higher lattices are not usually used because generation cost gets quite high.

The random function has a couple of competing requirements: It must not produce any obvious pattern, but it also must not vary every time we generate it, because we need the noise to be repeatable. The latter requirement is because otherwise the resulting textures will appear to flicker and move

across the surfaces they're applied to. While this may be desireable in some cases, we'd like to control the situation. To manage this, most noise systems pregenerate a table of random values and then hash into the table, where the hash is usually based on the lattice coordinates. We also want these random values to be bounded — the most common interval is $[-1, 1]$.

The most basic lattice noise is known as *value noise*. In this case, we generate random values at each lattice point and then interpolate between them. An alternative is Ken Perlin's original noise function [90] also known as *gradient noise*. In this case, the position at each lattice point is set to zero, but the tangent vector is randomized. This can be done generating a random point on a unit sphere, as we did above. Value noise tends to be lower frequency (more smooth) and gradient noise tends to be high frequency (more jaggy). Because of this, it's also common to combine them to create *value-gradient noise*.

To create interesting effects, we combine noise functions together. Often, we use the same noise function, but vary the spacing of the lattice. By doubling the frequency of lattice points we get what is called a new noise *octave*. This gives us a higher level of detail, which we can either use alone or combine with other octaves to get a more naturalistic effect: the lower octaves provide the broad strokes, while the higher octaves add the fiddly bits. For example, combining four octaves of gradient noise together gives us a *turbulence* function, which is very useful for producing cloud and marble effects. Usually the higher octaves are divided by their relative frequency before adding to help blend their effect into the lower-frequency base.

Let's look at a couple of examples using fragment shaders. Both of these are simplified from the OpenGL Shading Language text [99]. The first example generates a cloud texture on our object.

```
varying vec3 localPos;
void main()
{
    vec3 sky = vec3(0.0, 0.3, 0.8);
    vec3 cloud = vec3(0.8, 0.8, 0.8);
    float turb = (noise1(localPos) + noise1(2.0*localPos)*0.5
        + noise1(4.0*localPos)*0.25 + noise1(8.0*localPos)*0.125);
    vec3 color = mix(sky, cloud, turb);
    gl_FragColor = vec4(color, 1.0);
}
```

Here we see the turbulence calculation. We use our nontransformed position (sent via a varying variable from the vertex shader) as the hash into our noise function and scale it to get different frequencies. We're using 2.0 as our frequency increment here to show the ideal behavior, but it's usually recommended to use a nonintegral value to decrease some of the gridlike behavior often seen with lattice noise. Once the turbulence value is calculated,

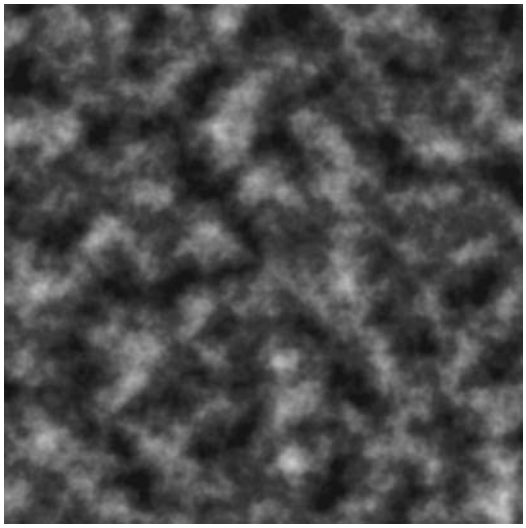


FIGURE 11.16 Sky texture generated using Perlin noise in a turbulence function.

we use it as a blending factor between our sky and cloud colors. Figure 11.16 shows the result.

We can do something similar to generate a marble texture. The base interpolant for the marble is the sine of the local y coordinate. We then perturb the base position by our turbulence to remove the regularity of the sine function as follows.

```
varying vec3 localPos;
void main()
{
    vec3 light = vec3(0.7, 0.7, 0.7);
    vec3 dark = vec3(0.0, 0.0, 0.0);
    float turb = (noise1(localPos) + noise1(2.0*localPos)*0.5
        + noise1(4.0*localPos)*0.25 + noise1(8.0*localPos)*0.125);
    float interp = sin(6.0*MCposition.y + 8.0*turb)*0.65;
    vec3 color = mix(light, dark, interp);
    gl_FragColor = vec4(color, 1.0);
}
```

Figure 11.17 shows the result.

For both of these cases, we have used the built-in noise function in GLSL. Similar noise functions are available in HLSL, Cg, and other shading languages. Whether you use this function or not depends on the speed of your graphics processing unit. In these fragment shaders we are doing four

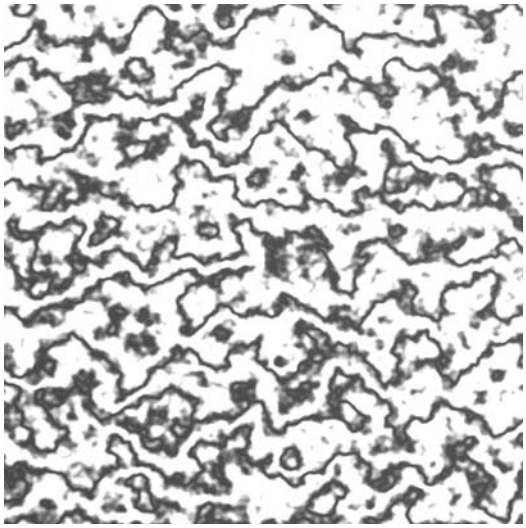


Figure 11.17 Marble texture generated using Perlin noise in a turbulence function.

function calls, which can get rather expensive. Because of this, graphics engineers often will generate a texture with different noise octave values in each color component and then do a lookup into that texture.

These examples give just a taste of what is available by making use of noise functions. Noise is used for generating wood textures, turbulence in fire texture, terrain, and many other cases. More detail on noise and other procedural generation can be found in Ebert et al. [29].

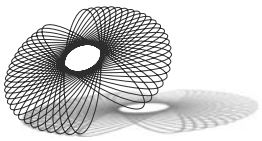
11.6 Chapter Summary

In this chapter we discussed some basic probability and statistics that will help us build our random processes. We used some of these statistic measures to create basic techniques for measuring random data. We also surveyed the most common random number generators, in particular, the linear congruential generator and the Mersenne Twister. Finally, we wrapped things up by looking at some examples of using these random number generators, from simulating arbitrary distributions to building turbulence functions for computer graphics.

Further reading in random numbers is wide and varied. Gonick and Smith [45] is a very approachable guide to probability and statistics; Grinstead and Snell [49] is recommended as a more thorough and formal, yet still readable, text. While slightly out of date now, the standard survey

of random number techniques can be found in Knuth [65]. A great deal of detail is given in this text to demonstrating the correctness of random algorithms and discussing techniques for measuring randomness. For those interested in unusual random distributions, particularly for graphics, Pharr and Humphries [91] is an excellent text. Finally, Ebert et al. [29] is the standard book for studying procedural algorithms.

This page intentionally left blank

Chapter 12

Intersection Testing

12.1 Introduction

In the previous chapters we have been primarily focused on manipulating and displaying our game objects in isolation. Whether we are rendering an object or animating it, we haven't been concerned with how it might be interacting with other objects in our scene. This is neither realistic nor interesting. For example, you are manipulating an object right now: this book. You can hold it in your hand, turn its pages, or drop it on the floor. In the latter case it stops reacting to you and starts reacting to the floor. If good game play derives from interesting interactions, then we need some way to detect when two game objects should be affecting one another and respond accordingly.

In this chapter we'll be concerned with a very straightforward question: How do we tell when two geometric entities are intersecting? This knowledge proves useful in many cases throughout a game engine. The most obvious is collision detection and response. Rather than have game objects pass through each other, we want them to push against each other and respond realistically. In the real world, this is a simple problem. Solid objects are solid; due to their physical properties, they just don't interpenetrate. But in the virtual world we have to create these constraints ourselves. Despite the fact that we have completely defined the geometry of our game objects, we still need to provide methods to detect when they interpenetrate. Only when we have a way to handle this can we write the code to perform the proper response.

Another time when we want to detect when two geometric entities interpenetrate is when we want to cast a ray and see what objects it intersects. One example of this we have seen already: detecting the object we've clicked

on by generating a pick ray from a screen-space mouse click, and determining the first object we hit with that ray. Another way this is used is in artificial intelligence (AI). In order to simulate whether one AI agent can see another, we cast a ray from the first to the second and see if it intersects any objects. If not, then we can say that the first agent's target is in sight.

We have also mentioned a third use of object intersection before: determining which objects are visible in a view frustum so that we can do quick visibility culling. If they interpenetrate or are inside the frustum, then we go ahead to the rendering step, otherwise they get skipped. This can considerably speed up our rendering.

Due to the variety of shapes and primitives used in a standard game engine, finding intersections between all of the cases can get quite complex; a single chapter is not enough to cover everything. Instead, we'll cover five basic objects, some methods for improving performance and accuracy, and directions for improvement. We will also briefly discuss how to use these methods in a simple collision detection system, and how we can apply similar techniques to our ray casting and frustum culling problems. Details on more complex systems can be found in the recommended reading at the end of the chapter.

12.2 Closest Point and Distance Tests

As we'll find, object intersection tests often can be described more easily in terms of a distance computation between two primitives, such as a point and a line. In particular, we'll often want to know if the distance between two primitives is less than some value, such as a radius. So, before we begin our discussion of determining intersections between bounding objects, we will cover a selection of useful methods for testing distances between certain geometric primitives.

Related to that topic is determining the closest points of approach between those same primitives; if we can find the closest points, the distance between the two primitives is the distance between those points. Because of this, we'll first consider closest point problems followed by how to calculate the distance between the same two primitives.

12.2.1 Closest Point on Line to Point

Source Code
Library
IvMath
Filename
IvLine3

Our first problem is illustrated in Figure 12.1: Given a point Q, and a line L defined by a point P and a vector $\mathbf{v}$, how do we find the point Q' on the line that is closest to Q? We approach this by examining the geometric

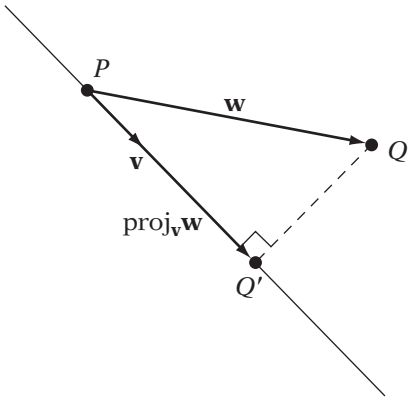


FIGURE 12.1 Closest point on a line.

relationships between the point and line. In particular, we notice that the dotted line segment between Q and Q' is orthogonal to the line. This line segment corresponds to a line of projection: To find Q', we need to project Q onto the line.

To do this, we begin by computing the difference vector $\mathbf{w}$ between Q and P, or $\mathbf{w} = Q - P$. Then we project this onto $\mathbf{v}$, to get the component of $\mathbf{w}$ that points along $\mathbf{v}$. Recall that this is

$$\text{proj}_{\mathbf{v}}\,\mathbf{w} = \frac{\mathbf{w} \cdot \mathbf{v}}{\|\mathbf{v}\|^2}\mathbf{v}$$

We add this to the line point P to get our projected point Q', or

$$Q' = P + \frac{\mathbf{w} \cdot \mathbf{v}}{\|\mathbf{v}\|^2}\mathbf{v}$$

The equivalent code is as follows:

```
IvVector3 IvLine3::ClosestPoint(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;
    float vsq = mDirection.Dot(mDirection);
    float proj = w.Dot(mDirection);

    return mOrigin + (proj/vsq)*mDirection;
}
```

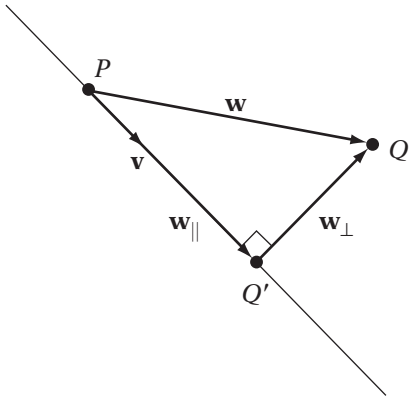


FIGURE 12.2 Computing distance from point to line, using a right triangle.

12.2.2 LINE–POINT DISTANCE

SOURCE CODE
LIBRARY
IvMath
FILENAME
IvLine3

As before, we're given a point Q and a line L defined by a point P and a vector $\mathbf{v}$. In this case, we want to find the distance between the point and the line. One way is to compute the closest point on the line and compute the distance between that and Q. A more direct approach is to use the Pythagorean theorem (Figure 12.2).

We note that $\mathbf{w} = Q - P$ can be represented as the sum of two vectors, one parallel to $\mathbf{v}(\mathbf{w}_{\parallel})$ and one perpendicular $(\mathbf{w}_{\perp})$. These form a right triangle, so from Pythagoras, $\|\mathbf{w}\|^2 = \|\mathbf{w}_{\parallel}\|^2 + \|\mathbf{w}_{\perp}\|^2$. We want to know the length of $\mathbf{w}_{\perp}$, so we can rewrite this as

$$\begin{aligned}
\|\mathbf{w}_{\perp}\|^2 &= \|\mathbf{w}\|^2 - \|\mathbf{w}_{\parallel}\|^2 \\
&= \mathbf{w} \cdot \mathbf{w} - \left\| \frac{\mathbf{w} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} \mathbf{v} \right\|^2 \\
&= \mathbf{w} \cdot \mathbf{w} - \left(\frac{\mathbf{w} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} \right)^2 \mathbf{v} \cdot \mathbf{v} \\
&= \mathbf{w} \cdot \mathbf{w} - \frac{(\mathbf{w} \cdot \mathbf{v})^2}{\mathbf{v} \cdot \mathbf{v}}
\end{aligned}$$

Taking the square root of both sides will give us the distance between the point and the line.

The equivalent code is as follows:

```
float IvLine3::DistanceSquared(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;
```

```
        float vsq = mDirection.Dot(mDirection);
        float wsq = w.Dot(w);
        float proj = w.Dot(mDirection);

        return wsq - proj*proj/vsq;
    }
```

Note that in this case we're computing the squared distance. In most cases we'll be using this to avoid computing a square root. Another optimization is possible if we can guarantee that $\mathbf{v}$ is normalized; in that case, we can avoid calculating and dividing by $\mathbf{v} \cdot \mathbf{v}$, since its value is 1.

12.2.3 Closest Point on Line Segment to Point

Source Code
Library
IvMath
Filename
IvLineSegment3

Recall that a line segment can be defined as the convex combination of two points P_0 and P_1, or

$$S(t) = (1 - t)P_0 + tP_1$$

where $0 \leq t \leq 1$. We can rewrite this as

$$S(t) = P_0 + t(P_1 - P_0)$$

or

$$S(t) = P + t\,\mathbf{v}$$

where as before $0 \leq t \leq 1$. In this case, $\mathbf{v}$ should not be normalized, as its length is the length of our line segment, and the endpoints are P and $P + \mathbf{v}$.

In the problem of finding the closest point on a line, we computed the projection of the point onto the line. Doing the same for a line segment gives us three cases (Figure 12.3). In the first case, the result of projecting Q_0 lies outside the segment but closest to P_0. In the second case, the result of projecting Q_1 lies outside the segment but closest to P_1. In the third case, the projected Q_2 lies on the segment, and we can use the same projection calculations that we used with a line.

To determine which case we're in, we begin by noting that

$$t = \frac{\mathbf{w} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}$$

is acting as our parameter t for the projected point, where again $\mathbf{w} = Q - P$. If $t < 0$, then the projected point lies beyond P_0, and the closest point is P_0. Similarly, if $t > 1$, then the closest point is P_1.

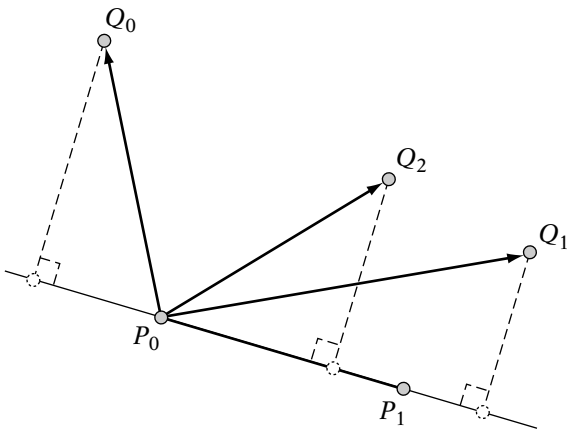


FIGURE 12.3 Three cases when projecting a point onto line segment.

Testing t directly requires a floating-point division. By modifying our test we can defer the division to be performed only when we truly need it; that is, when the point lies on the segment. Since $\mathbf{v} \cdot \mathbf{v} > 0$, then $\mathbf{w} \cdot \mathbf{v} < 0$ in order for $t < 0$. And in order for $t > 1$, then $\mathbf{w} \cdot \mathbf{v} > \mathbf{v} \cdot \mathbf{v}$.

The equivalent code is as follows:

```
IvVector3 IvLineSegment3::ClosestPoint(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;

    float proj = w.Dot(mDirection);
    if ( proj <= 0 )
        return mOrigin;
    else
    {
        float vsq = mDirection.Dot(mDirection);
        if ( proj >= vsq )
            return mOrigin + mDirection;
        else
            return mOrigin + (proj/vsq)*mDirection;
    }
}
```

12.2.4 Line Segment–Point Distance

SOURCE CODE
LIBRARY
IvMath
FILENAME
IvLineSegment3

As with lines, we can compute the distance to the line segment by computing the distance to the closest point on the line segment. If we recall, there

are three cases: the closest point is P_0, P_1, or a point somewhere else on the segment, which we'll calculate.

If the closest point is P_0, then we can compute the distance as $\|Q - P_0\|$. Since $\mathbf{w} = Q - P_0$, then the squared distance is equal to $\mathbf{w} \cdot \mathbf{w}$.

If the closest point is P_1, then the squared distance is $(Q - P_1) \cdot (Q - P_1)$. However, we're representing our endpoint as $P_1 = P_0 + \mathbf{v}$, so this becomes $(Q - P_0 - \mathbf{v}) \cdot (Q - P_0 - \mathbf{v})$. We can rewrite this as

$$\begin{aligned} \text{distsq}(Q, P_1) &= ((Q - P_0) - \mathbf{v}) \cdot ((Q - P_0) - \mathbf{v}) \\ &= (\mathbf{w} - \mathbf{v}) \cdot (\mathbf{w} - \mathbf{v}) \\ &= \mathbf{w} \cdot \mathbf{w} - 2\mathbf{w} \cdot \mathbf{v} + \mathbf{v} \cdot \mathbf{v} \end{aligned}$$

We've already calculated most of these dot products when determining whether we're closest to P_1, so all we need to compute is $\mathbf{w} \cdot \mathbf{w}$ and add. If the closest point lies elsewhere on the segment, then we use the line distance calculation just given.

The final code is as follows:

```
float IvLineSegment3::DistanceSquared(const IvVector3& point)
{
    IvVector3 w = point - mOrigin;

    float proj = w.Dot(mDirection);
    if ( proj <= 0 )
    {
        return w.Dot(w);
    }
    else
    {
        float vsq = mDirection.Dot(mDirection);
        if ( proj >= vsq )
        {
            return w.Dot(w) - 2.0f*proj + vsq;
        }
        else
        {
            return w.Dot(w) - proj*proj/vsq;
        }
    }
}
```

12.2.5 Closest Points Between Two Lines

SOURCE CODE
LIBRARY
IvMath
FILENAME
IvLine3

Sunday [107] provides the following construction for finding the closest points between two lines. Note that in this case there are two closest points, one on each line, since there are two degrees of freedom. The situation is shown in Figure 12.4. Line L_1 is described by the point P_0 and the vector $\mathbf{u}$. Correspondingly, line L_2 is described by the point Q_0 and the vector $\mathbf{v}$, or

$$L_1(s) = P_0 + s\,\mathbf{u}$$
$$L_2(t) = Q_0 + t\,\mathbf{v}$$

Vectors $\mathbf{u}$ and $\mathbf{v}$ are not necessarily normalized.

We'll define the two closest points that we're looking for as lying at parameters s_c and t_c on the lines, and call them $L_1(s_c)$ and $L_2(t_c)$, respectively. We'll refer to the vector from $L_2(t_c)$ to $L_1(s_c)$ as $\mathbf{w}_c$.

Expanding $\mathbf{w}_c$, we have

$$\begin{aligned}\mathbf{w}_c &= L_1(s_c) - L_2(t_c)\\ &= P_0 + s_c\,\mathbf{u} - Q_0 - t_c\,\mathbf{v}\\ &= (P_0 - Q_0) + s_c\,\mathbf{u} - t_c\,\mathbf{v}\end{aligned}$$

We'll use $\mathbf{w}_0$ to represent the difference vector $P_0 - Q_0$, so

$$\mathbf{w}_c = \mathbf{w}_0 + s_c\,\mathbf{u} - t_c\,\mathbf{v} \tag{12.1}$$

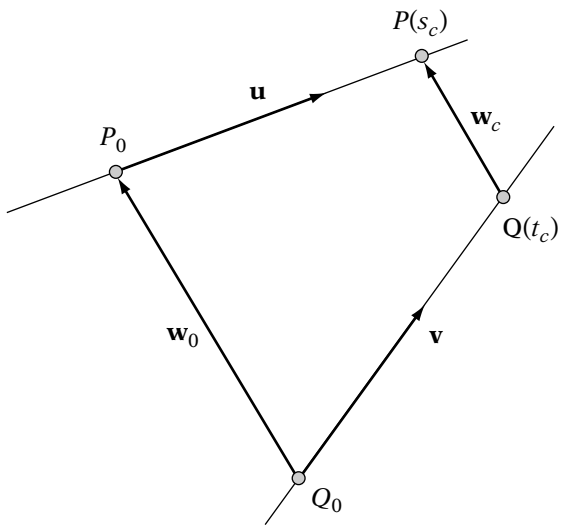


Figure 12.4 Finding the closest points between two lines.

In order for $\mathbf{w}_c$ to represent the vector of closest distance, it needs to be perpendicular to both L_1 and L_2. This means that

$$\mathbf{w}_c \cdot \mathbf{u} = 0$$
$$\mathbf{w}_c \cdot \mathbf{v} = 0$$

Substituting in equation 12.1 and expanding, we get

$$0 = \mathbf{w}_0 \cdot \mathbf{u} + s_c \mathbf{u} \cdot \mathbf{u} - t_c \mathbf{u} \cdot \mathbf{v} \tag{12.2}$$
$$0 = \mathbf{w}_0 \cdot \mathbf{v} + s_c \mathbf{u} \cdot \mathbf{v} - t_c \mathbf{v} \cdot \mathbf{v} \tag{12.3}$$

We have two equations and two unknowns s_c and t_c, so we can solve for this system of equations. Doing so, we get the result that

$$s_c = \frac{be - cd}{ac - b^2} \tag{12.4}$$
$$t_c = \frac{ae - bd}{ac - b^2} \tag{12.5}$$

where

$$a = \mathbf{u} \cdot \mathbf{u}$$
$$b = \mathbf{u} \cdot \mathbf{v}$$
$$c = \mathbf{v} \cdot \mathbf{v}$$
$$d = \mathbf{u} \cdot \mathbf{w}_0$$
$$e = \mathbf{v} \cdot \mathbf{w}_0$$

There is one case where we need to be careful. If the two lines are parallel, then $\mathbf{u}$ and $\mathbf{v}$ are parallel, so $|\mathbf{u} \cdot \mathbf{v}| = \|\mathbf{u}\|\|\mathbf{v}\|$. Then the denominator $ac - b^2$ equals

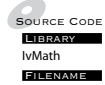
$$\begin{aligned} ac - b^2 &= (\mathbf{u} \cdot \mathbf{u})(\mathbf{v} \cdot \mathbf{v}) - (\mathbf{u} \cdot \mathbf{v})^2 \\ &= \|\mathbf{u}\|^2\|\mathbf{v}\|^2 - (\|\mathbf{u}\|\|\mathbf{v}\|)^2 \\ &= 0 \end{aligned}$$

This leads to a division by 0. The problem is that there are an infinite number of pairs of closest points spaced along each line. In this case, we'll just find the closest point Q' on L_2 to the origin P_0 of line L_1 and return P_0 and Q'.

```
void ClosestPoints( IvVector3& point1,
                    IvVector3& point2,
                    const IvLine3& line1,
                    const IvLine3& line2 )
{
    IvVector3 w0 = line1.mOrigin - line2.mOrigin;
    float a = line1.mDirection.Dot( line1.mDirection );
    float b = line1.mDirection.Dot( line2.mDirection );
    float c = line2.mDirection.Dot( line2.mDirection );
    float d = line1.mDirection.Dot( w0 );
    float e = line2.mDirection.Dot( w0 );

    float denom = a*c - b*b;
    if ( ::IsZero(denom) )
    {
        point1 = line1.mOrigin;
        point2 = line2.mOrigin + (e/c)*line2.mDirection;
    }
    else
    {
        point1 = line1.mOrigin + ((b*e - c*d)/denom)*line1.mDirection;
        point2 = line2.mOrigin + ((a*e - b*d)/denom)*line2.mDirection;
    }
}
```

12.2.6 Line–Line Distance



From the calculation of closest points between two lines, we know that $\mathbf{w}_c$ is the vector of closest distance. Therefore, its length equals the distance between the two lines. Rather than compute the closest points directly, we can substitute the values of s_c and t_c into equation 12.1 and compute the length of $\mathbf{w}_c$. As before, to avoid the square root, we can use $\|\mathbf{w}_c\|^2 = \mathbf{w}_c \cdot \mathbf{w}_c$ instead.

The code is as follows:

```
float DistanceSquared( const IvLine3& line1, const IvLine3& line2 )
{
    // compute parameters
    IvVector3 w0 = line1.mOrigin - line2.mOrigin;
    float a = line1.mDirection.Dot( line1.mDirection );
    float b = line1.mDirection.Dot( line2.mDirection );
    float c = line2.mDirection.Dot( line2.mDirection );
    float d = line1.mDirection.Dot( w0 );
    float e = line2.mDirection.Dot( w0 );
```

```
        float denom = a*c - b*b;
        // if lines parallel
        if ( ::IsZero(denom) )
        {
            IvVector3 wc = w0 - (e/c)*line2.mDirection;
            return wc.Dot(wc);
        }
        // otherwise
        else
        {
              IvVector3 wc = w0 + ((b*e - c*d)/denom)*line1.mDirection
                                - ((a*e - b*d)/denom)*line2.mDirection;
              return wc.Dot(wc);
        }
    }
```

12.2.7 Closest Points Between Two Line Segments

Finding the closest points between two line segments follows from finding the closest points between two lines. We compute s_c and t_c, as we've done, but then need to clamp the results to the ranges of s and t defined by the endpoints of the two line segments. As before, we'll define our line segments as starting at the source point of the line and ending at that source point plus the line vector. So for line L_1, the two points are P_0 and $P_0 + \mathbf{u}$, and for line L_2, the two points are Q_0 and $Q_0 + \mathbf{v}$. This gives us parameters 0 and 1 for the locations of the two endpoints. If our results s_c and t_c lie between the values 0 and 1, then our closest points lie on the two segments, and we're done.

Otherwise, we need to clamp our parameters to each of the endpoint parameters and try again. To see how to do that, let's take a look at the $s = 0$ endpoint. Remember that what we want to do is find the smallest possible distance between the two points while not sliding off the end of the segment; namely, we want to minimize the length of $\mathbf{w}_c$ while maintaining $s = 0$. Since length is always increasing, we'll use $\|\mathbf{w}_c\|^2$, which will be much easier to minimize. Remember that

$$\mathbf{w}_c = \mathbf{w}_0 + s_c\,\mathbf{u} - t_c\,\mathbf{v}$$

Since we're clamping s_c to 0, this becomes

$$\mathbf{w}_c = \mathbf{w}_0 - t_c\,\mathbf{v}$$

Therefore, for this endpoint we try to find the minimum value for

$$\mathbf{w}_c \cdot \mathbf{w}_c = (\mathbf{w}_0 - t_c \mathbf{v}) \cdot (\mathbf{w}_0 - t_c \mathbf{v}) \tag{12.6}$$

To do this, we return to calculus. To find a minimum value (in this case, there is only one) for a function, we find a place where the derivative is 0. Taking the derivative of equation 12.6 in terms of t_c, we get the result

$$0 = -2\mathbf{v} \cdot (\mathbf{w}_0 - t_c \mathbf{v})$$

Solving for t_c, we get

$$t_c = \frac{\mathbf{v} \cdot \mathbf{w}_0}{\mathbf{v} \cdot \mathbf{v}} \tag{12.7}$$

So, for the fixed point on line L_1 at $s = 0$, this gives us the parameter of the closest point on line L_2. As we can see, this is equivalent to computing the closest point between a line and a point, where the line is L_2 and the point is P_0.

For the $s = 1$ endpoint, we follow a similar process. Our minimization function is

$$\mathbf{w}_c \cdot \mathbf{w}_c = (\mathbf{w}_0 + \mathbf{u} - t_c \mathbf{v}) \cdot (\mathbf{w}_0 + \mathbf{u} - t_c \mathbf{v}) \tag{12.8}$$

The corresponding zero derivative function is

$$0 = -2\mathbf{v} \cdot (\mathbf{w}_0 + \mathbf{u} - t_c \mathbf{v})$$

Solving for t_c gives us

$$t_c = \frac{\mathbf{v} \cdot \mathbf{w}_0 + \mathbf{u} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}}$$

Again, this is equivalent to computing the closest point between a line and a point, where the line is L_2 and the point is $P_0 + \mathbf{v}$. The solutions for s_c when clamping to $t = 0$ or $t = 1$ are similar.

One nice thing about these functions is that they use the *a* through *e* values that we've already calculated for the basic line–line distance calculation. So, equation 12.7 becomes

$$t_c = \frac{e}{c}$$

So, which endpoints do we check? Well, if the parameter s_c is less than 0, then the closest segment point to line L_2 will be the $s = 0$ endpoint. And if s_c

is greater than 1, then the closest segment point will be at $s = 1$. Choosing one or the other, we resolve for t_c and check that it lies between 0 and 1. If not, we perform the same process to clamp t_c to either the $t = 0$ or $t = 1$ endpoint and recalculate s_c accordingly (with some minor adjustments to ensure that we keep s_c within 0 and 1).

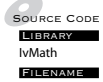
Once again, there is a trick we can do to avoid multiple floating-point divisions. Instead of computing, say, s_c directly and testing against 0 and 1, we can compute the numerator s_N and denominator s_D. The initial s_D is always greater than zero, so we know that if s_N is less than zero, s_c is less than zero and we clamp to $s = 0$ accordingly. Similarly, if s_N is greater than s_D, we know that $s_c > 1$, and we clamp to $s = 1$. The same can be done for the t values. Using this, we can recalculate the numerator and denominator when necessary, and do the floating-point divides only after all the clamping has been done.

For example, the following code snippet calculates the s values:

```
// clamp s_c to 0
if (sN < 0.0f)
{
    sN = 0.0f;
    tN = e;
    tD = c;
}
// clamp s_c to 1
else if (sN > sD)
{
    sN = sD;
    tN = e + b;
    tD = c;
}
```

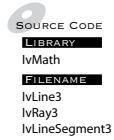
The full code is too long to contain here, but can be found on the demo CD-ROM.

12.2.8 Line Segment–Line Segment Distance



Finding the segment-to-segment squared distance is similar to line-to-line distance: We follow the procedure for closest points between line segments, calculate $\mathbf{w}_c$ directly from the final s_c and t_c, and then compute its length. The full code can be found on the CD-ROM in the `IvLineSegment3` friend function `DistanceSquared()`.

12.2.9 General Linear Components



Testing ray versus ray or line versus line segments is actually a simplification of the segment–segment closest point and distance determination. Instead of clamping against both components, we need only clamp against those endpoints that are necessary. So for example, if we treat $P_0 + s\mathbf{u}$ as the parameterization of a line segment, and $Q_0 + t\mathbf{v}$ as a line, then we need only to ensure that s_c is between 0 and 1, clamp to the appropriate endpoint, and adjust t_c accordingly. Similarly, if we're working with rays, we need only to clamp s_c or t_c to 0.

Implementations of these algorithms can be found in the appropriate classes.

12.3 Object Intersection

Now that we've covered some methods for measuring distance between primitives, we can talk about object intersection. The most direct, and naive, approach to determine whether two objects are intersecting is to work directly from raw object data. We could start with a triangle in object A and a triangle in object B and see if they are intersecting. Then we move to the next triangle in object A and test again. While ultimately this may work (the exception is if one object is inside the other), it will take a while to do and most of the time performing all those tests isn't even necessary. Take the two objects in Figure 12.5. They are clearly not intersecting — we can tell that in an instant. But our minds are not considering each object as a collection of lines and doing individual tests. Rather, we are comparing them as a whole, as two rough blobs, and determining that the blobs aren't intersecting. By using a similar process in our intersection routines, we can save ourselves a lot of time.

For instance, suppose we surround each object with a sphere (Figure 12.6). We can begin by testing for intersection between the spheres. If the two spheres aren't intersecting, we know the objects aren't either. If the spheres are intersecting, we can try comparing another simplified version of our object — say, two boxes. The boxes fit the shape of our objects better but are still a simpler test than our full triangle–triangle comparison. If the boxes intersect, only then do we perform our complex collision detection routine.

This technique of using simplified objects to test intersections before performing more expensive operations is commonly used in game engines, and is necessary to get collision detection and other intersection-based systems running in real time. The simplified objects are known as *bounding objects* and are named specifically after the basic primitives we used to approximate the object: bounding spheres and bounding boxes. In games, we can often

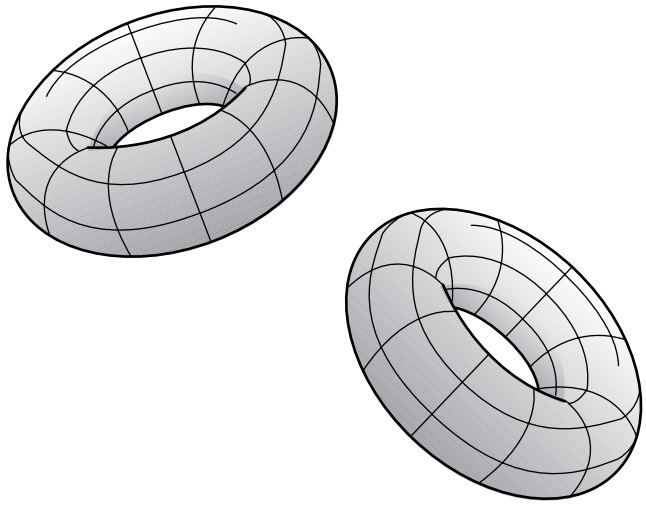


FIGURE 12.5 Nonintersecting objects.

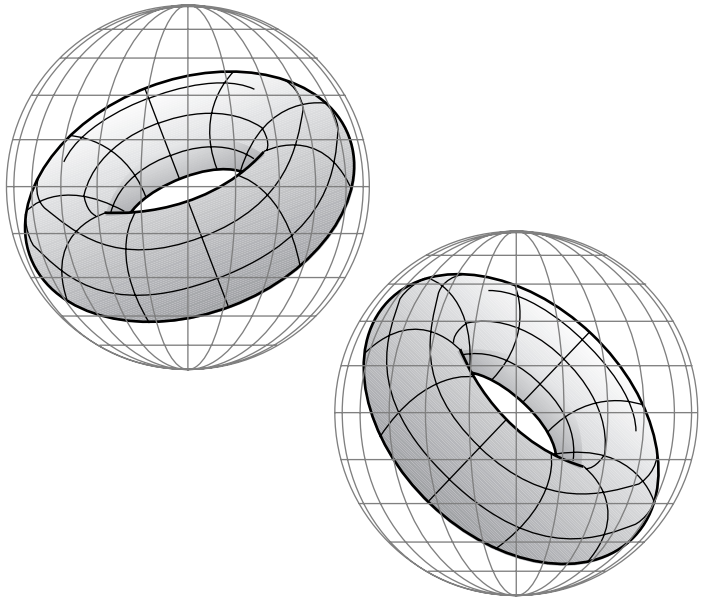


FIGURE 12.6 Nonintersecting objects with bounding sphere.

get away with ignoring the underlying geometry completely and only using bounding objects to determine intersections. For example, when handling collisions in this way, either the action happens so fast that we don't notice any overlapping objects or objects reacting to collision when they appear

separated, or the error is so slight that it doesn't matter. In any case, choosing the side of making the simulation run faster for a better play experience is usually a good decision.

One thing to note with the following algorithms is that their performance is often dependent on the platform that they are run on. For example, many consoles don't have predictive branching, so conditionals are quite slow. So, on such a platform, an algorithm that calculates unnecessary data may actually turn out to be faster than one that attempts to avoid this using if-then-else clauses. Even on relatively similar architectures there can be surprising differences in relative performance. This is shown strikingly by Löfstedt and Akenine-Möller [69].

To keep things concise, we have chosen a few algorithms that are commonly used and are relatively fast on a broad variety of architectures. Other books are more detailed, covering many different polytopes (the 3D equivalent of polygons) and interactions between all sorts of bounding objects. In our case, we'll focus on a few simple shapes, beginning with the simplest objects and moving on to the most complex, or most expensive, to compute. However, the reader should be aware of the issues above and may need to explore alternatives for his or her particular application.

Within each section we'll only consider three cases of intersection. We'll first look at intersections between objects of the same bounding type, which is useful in collision detection. Second, we'll cover intersections between a ray and the particular bounding object, which we'll need for picking and visibility testing for AI. Finally, we'll discuss how to determine intersection between a plane and the bounding object, which can be used for both culling against frustum planes and collisions with essential planar objects like walls. In all cases, we aren't concerned with the exact point of intersection, just whether the items intersect.

12.3.1 Spheres

Definition

Source Code
LIBRARY
IvCollision
FILENAME
IvBounding Sphere

The simplest possible bounding object is a sphere. It also has the most compact representation: a center point C and a radius r (Figure 12.7). When bounding a rigid object, a sphere is also independent of the object's orientation. This allows us to update a sphere quickly — when an object moves, we need only to update the sphere's position. If the object is scaled, we can scale the radius accordingly. The combination of low memory usage, fast update time, and fast intersection tests makes bounding spheres a first choice in any real-time system.

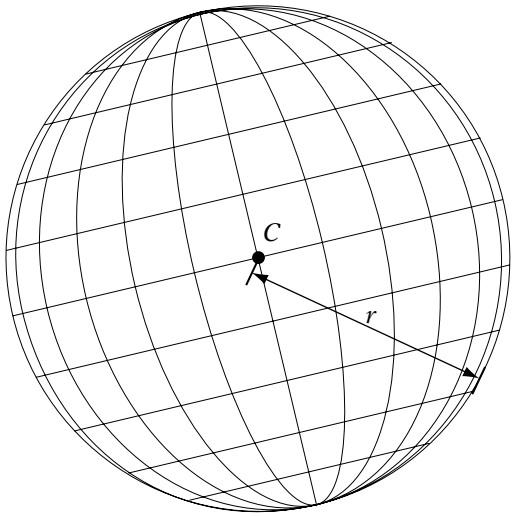


FIGURE 12.7 Bounding sphere.

The surface of the sphere is defined as all points P such that the length of the vector from C to P is equal to the radius:

$$\sqrt{(P_x - C_x)^2 + (P_y - C_y)^2 + (P_z - C_z)^2} = r$$

or

$$\sqrt{(P - C) \cdot (P - C)} = r$$

Ideally, we'll want to choose the smallest possible sphere that encompasses the entire object. Too small a sphere, and we may skip two objects that are actually intersecting. Too large, and we'll be unnecessarily performing our more expensive tests for objects that are clearly separate. Unfortunately, the most obvious methods for choosing a bounding sphere will not always generate as tight a fit as we might like.

One such method is to take the local origin of the object as our center C, and compute r by taking the maximum distance from that to all the vertices in the object. There are many problems with this. The most common is that the local origin could be considerably offset from the most desirable center point for the object (Figure 12.8(a)). This could happen if you have a character whose origin is at its feet, so it can be placed on the ground properly. An alternate but equivalent situation is where the origin is at a reasonable center

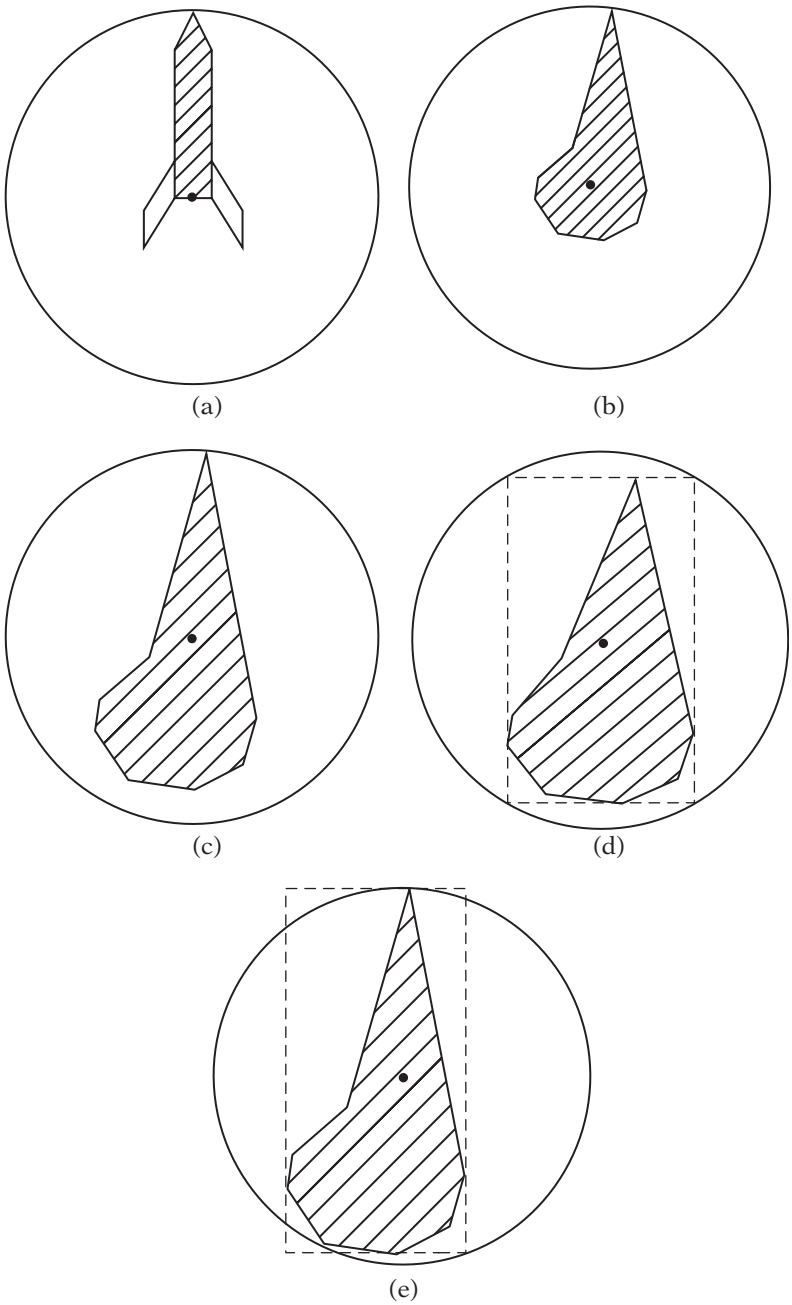


FIGURE 12.8 (a) Bounding sphere, offset origin; (b) bounding sphere, outlying point; (c) bounding sphere, using centroid, object vertices; (d) bounding sphere, using box center, box vertices; and (e) bounding sphere, using box center, object vertices.

point for the majority of the object's vertices, but there are one or two outlying vertices that cause problems (Figure 12.8b).

Eberly [25] provides a number of methods for finding a better fit. One is to average all the vertex locations to get the centroid and use that as our center. This works well for the case of a noncentered origin, but still is a problem for an object with outlying points (Figure 12.8c). The reason is that the majority of the points lie within a small area and thus weight the centroid in that direction, pulling it away from the extrema.

We could also take an axis-aligned bounding box in the object's local space and use its endpoints to compute our sphere position and radius (Figure 12.8(d)). This tends to center the sphere better but leads to a looser fit. A compromise method uses the center of the bounding box as our sphere position, and computes the radius as the maximum distance from the center to our points. This gives a slightly better result (Figure 12.8(e)). The code for this last method is as follows.

```
void
IvBoundingSphere::Set( const IvPoint3* points, unsigned int numPoints )
{
  // compute minimal and maximal bounds
  IvVector3 min(points[0]), max(points[0]);
  for ( unsigned int i = 1; i < numPoints; ++i )
  {
    if (points[i].x < min.x)
      min.x = points[i].x;
    else if (points[i].x > max.x )
      max.x = points[i].x;
    if (points[i].y < min.y)
      min.y = points[i].y;
    else if (points[i].y > max.y )
      max.y = points[i].y;
    if (points[i].z < min.z)
      min.z = points[i].z;
    else if (points[i].z > max.z )
      max.z = points[i].z;
  }

  // compute center and radius
  mCenter = 0.5f*(min + max);
  float maxDistance = ::DistanceSquared( mCenter, points[0] );
```

```
  for ( unsigned int i = 1; i < numPoints; ++i )
  {
    float dist = ::DistanceSquared( mCenter, points[i] );
    if (dist > maxDistance)
      maxDistance = dist;
  }
  mRadius = ::IvSqrt( maxDistance );
}
```

It should be noted that none of these methods is guaranteed to find the smallest bounding sphere. The standard algorithm for this is by Welzl [118], who showed that linear programming can be used to find the optimally smallest sphere surrounding a set of points. Two implementations are readily available online: one by Bernd Gaertner is provided under the GNU General Public License; another by Dave Eberly is at *www.magic-software.com*.

While we don't want to be cavalier about using ridiculously large bounding spheres, in some cases having the tightest possible fit isn't that much of an issue. Our objects will not be generally spherical, and so we'll be using something more complex for our final intersection test. As long as our spheres are reasonably close to a good fit, they will act to cull a great number of obvious cases, which is all we can ask for.

Sphere–Sphere Intersection

Determining whether two spheres are intersecting is as simple as their representation. We need only to determine whether the distance between their centers is less than the sum of their two radii (Figure 12.9), or

$$\sqrt{(C_1 - C_2) \cdot (C_1 - C_2)} <= r_1 + r_2 \tag{12.9}$$

The square root operation is expensive, and in any case, it is unnecessary. Since we're not looking for the absolute difference, just a relation, we can use

$$(C_1 - C_2) \cdot (C_1 - C_2) <= (r_1 + r_2)^2 \tag{12.10}$$

As promised, this gives us an extremely cheap test for culling large numbers of intersections. This is why bounding spheres are used everywhere in computer graphics and simulation; we perform an initial fast check with a bounding sphere first before even considering the more complex cases.

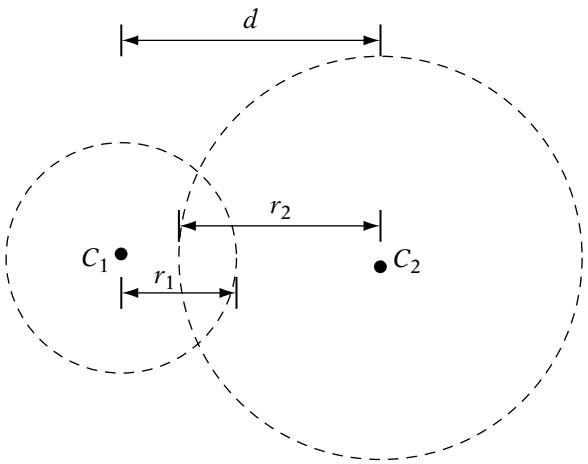


FIGURE 12.9 Sphere–sphere intersection.

The code is as follows:

```
bool
IvBoundingSphere::Intersect( const IvBoundingSphere& other )
{
    IvVector3 centerDiff = mCenter - other.mCenter;
    float radiusSum = mRadius + other.mRadius;
    return ( centerDiff.Dot(centerDiff) <= radiusSum*radiusSum );
}
```

Sphere–Ray Intersection

Intersection between a sphere and a ray is nearly as simple. Instead of testing two centers and comparing the distance with the sum of two radii, we test the distance between a single sphere center and a ray. If the distance is less than or equal to the sphere's radius, then the ray intersects the sphere (Figure 12.10).

We can use the line–point distance measurement described as the basis for this test. The code is as follows (it assumes an initial nonzero, nonnormalized **v**):

```
bool
IvBoundingSphere::Intersect( const IvRay3& ray )
{
    // compute intermediate values
    IvVector3 w = mCenter - ray.mOrigin;
```

```
        float wsq = w.Dot(w);
        float proj = w.Dot(ray.mDirection);
        float rsq = mRadius*mRadius;

        // if sphere behind ray, no intersection
        if ( proj < 0.0f && wsq > rsq )
           return false;

        float vsq = ray.mDirection.Dot(ray.mDirection);

        // test length of difference vs. radius
        return ( vsq*wsq - proj*proj <= vsq*mRadius*mRadius );
    }
```

An additional check has been added since we're using a ray. If the sphere lies behind the origin of the ray, then there is no intersection. This is true if the angle between the difference vector **w** and the line direction is greater than 90 degrees (`proj < 0.0f`) and the line origin lies outside of the sphere (`wsq > rsq`).

We also remove the need for a floating-point divide by multiplying through by `vsq`. This adds two multiplications, but this still should be faster on most floating-point processors. As before, if we can guarantee that the ray direction vector is normalized, then we can remove the need for `vsq` altogether.

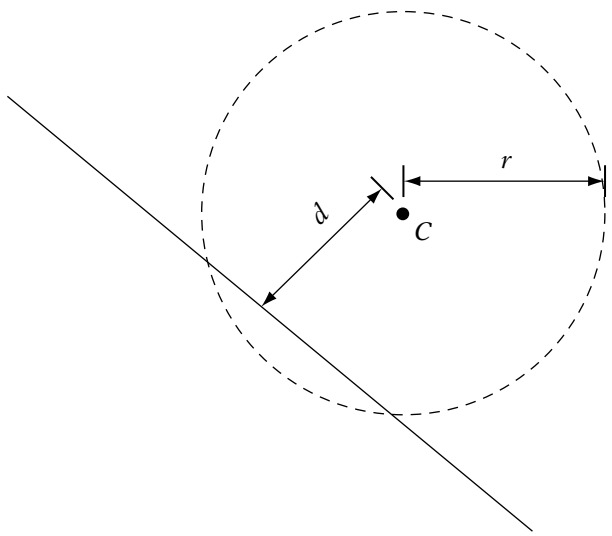


Figure 12.10 Line–sphere intersection.

Sphere–Plane Intersection

Testing whether a sphere lies entirely on one side of a plane can be done quite efficiently. Recall that we can determine the distance between a point and such a plane by taking the absolute value of the result of the plane equation. If the result is positive and the distance is greater than the radius, then the sphere lies on the inside of the plane. If the result is negative, and the distance is greater than the sphere's radius, then the sphere lies outside of the plane. Otherwise, the sphere intersects the plane.

The code for this test is as follows:

```
float
IvBoundingSphere::Classify( const IvPlane& plane )
{
    float distance = plane.test(mCenter):
    if ( distance > radius)
    {
        return distance-radius;
    }
    else if ( distance < -radius )
    {
        return distance+radius;
    }
    else
    {
        return 0.0f;
    }
}
```

Here we're returning a signed distance, like the standard plane test. If the sphere intersects, we return zero. Otherwise, we return the signed distance minus the signed distance of the radius.

12.3.2 Axis-Aligned Bounding Boxes

Definition

Spheres work well as either cheap culling objects or as bounding objects for a small class of models (i.e., if you're tossing grenades or writing a billiards game). For more angular objects, we need a better-fitting bounding surface. One possibility is the bounding box. Just like the bounding sphere, the ideal bounding box is the smallest possible box that encloses an object.

The first type we'll consider is the AABB, or axis-aligned bounding box, so called because the box edges are aligned to the *world* axes. This makes representation of the box simple: We use two points, one each for the minimum and maximum *xyz* positions (Figure 12.11). When the object is translated, to update the box we translate the minimum and maximum points. Similarly, if the object is scaled, we scale the two points relative to the box center. However, because the box is aligned to the world axes, any rotation of the object means that we have to recalculate the minimum and maximum points from the object vertices' new positions in world space.

The other disadvantage AABBs have is that in many cases, like spheres, they still aren't a very close fit to the object they are trying to approximate (Figure 12.12). And for rounded objects like submarines or organic objects like humans, the fact that they have corners is a disadvantage as well. However,

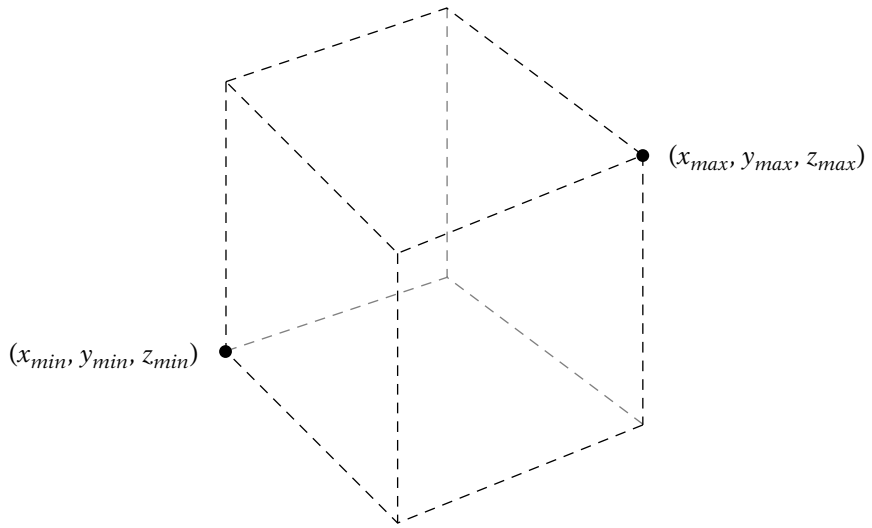


FIGURE 12.11 Axis-aligned bounding box.

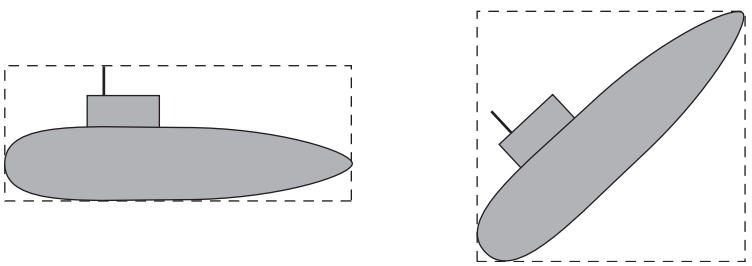


FIGURE 12.12 Fitting axis-aligned bounding box.

they are relatively cheap to compute and cheap to test as well, so they continue to prove useful.

One advantage that world axis-aligned boxes have over a box oriented to the object's local space is that we need only recompute them once per frame, and then we can compare them directly without further transformation, since they are all in the same coordinate frame. So, while AABBs have a high per-frame overhead (since we have to recalculate them each time an object reorients), they are *extremely* cheap to test against one another. As we'll see, there is a lot more overhead for determining intersection between oriented boxes. Oriented boxes are generally cheap per frame (they move with the transforms of the object), but are more expensive to test against one another.

To compute an AABB, we first transform the object into world space. Then we set the minimum and maximum points to be equal to the first point (in world space, remember) in the object. Starting with the second point, we compare the *xyz* values of each point with those in the minimum and maximum. If any coordinate is less than that in the minimum, set the minimum coordinate to that value, and the same for the maximum, except use greater than. When done, this will give you the axis-aligned extrema for your box.

```
void
IvAABB::Set( const IvPoint3* points, unsigned int numPoints )
{
    ASSERT( points );

    // compute minimal and maximal bounds
    mMinima.Set(points[0]);
    mMaxima.Set(points[0]);
    for ( unsigned int i = 1; i < numPoints; ++i )
    {
        if (points[i].x < mMinima.x)
            mMinima.x = points[i].x;
        else if (points[i].x > mMaxima.x )
            mMaxima.x = points[i].x;
        if (points[i].y < mMinima.y)
            mMinima.y = points[i].y;
        else if (points[i].y > mMaxima.y )
            mMaxima.y = points[i].y;
        if (points[i].z < mMinima.z)
            mMinima.z = points[i].z;
        else if (points[i].z > mMaxima.z )
            mMaxima.z = points[i].z;
    }
}
```

AABB–AABB Intersection

In order to understand how we find intersections between two axis-aligned boxes, we introduce the notion of a *separating plane*. The general idea is this: We check the boxes in each of the coordinate directions in world space. If we can find a plane that separates the two boxes in any of the coordinate directions, then the two boxes are not intersecting. If we fail all three separating plane tests, then they are intersecting and we handle it appropriately.

Let's look at the process of finding a separating plane between two boxes in the x direction. Since the boxes are axis aligned, this becomes a one-dimensional (1D) problem on a number line. The minimum and maximum values of the two boxes become the extrema of two intervals on the line. If the two intervals are separate, then there is a separating plane and the two boxes are separate along the x direction. This is the case only if the maximum value of one interval is less than the minimum value of the other interval (Figure 12.13). Expressing this for all three axes:

```
bool
IvAABB::Intersect( const IvAABB& other )
{
  // if separated in x direction
  if (mMinima.x > other.mMaxima.x || other.mMinima.x > mMaxima.x )
    return false;

  // if separated in y direction
  if (mMinima.y > other.mMaxima.y || other.mMinima.y > mMaxima.y )
    return false;

  // if separated in z direction
  if (mMinima.z > other.mMaxima.z || other.mMinima.z > mMaxima.z )
    return false;

  // no separation, must be intersecting
  return true;
}
```

Examining this code makes another advantage of AABBs clear. If we're using three-dimensional (3D) objects in an essentially two-dimensional (2D) game, we can ignore the z-axis and so save a step in our computations. This is not always possible with boxes aligned to the local axes of an object.

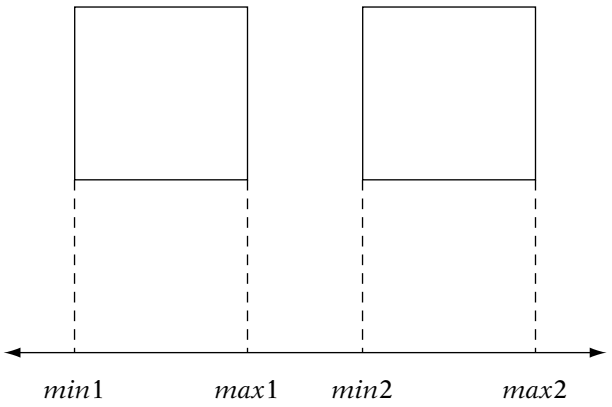


FIGURE 12.13 Axis-aligned box–box separation test.

AABB–Ray Intersection

Determining intersection between a ray and an axis-aligned box is similar to determining intersection between two boxes. We check one axis direction at a time as before, except that in this case, there is a little more interaction between steps.

Figure 12.14 shows a 2D cross section of the situation. The ray R shown intersects the minimum and maximum x planes of the box at $R(s_x)$ and $R(t_x)$, respectively, and the minimum and maximum y planes at $R(s_y)$ and $R(t_y)$. Instead of testing for extrema overlaps in the box axes directions, we'll test whether there is overlap between the line segment from $R(s_x)$ to $R(t_x)$, and the line segment from $R(s_y)$ to $R(t_y)$. This is the same as testing whether the intervals of the line parameters $[s_x, t_x]$ and $[s_y, t_y]$ overlap.

If the ray misses the box, as in the figure, then the $[s_x, t_x]$ interval doesn't overlap the $[s_y, t_y]$ interval, just like the preceding box–box intersection. So, if there's no overlap (if $t_x < s_y$, or vice versa), then there's no intersection, and we stop. If they do overlap, then we test that overlap interval against the z intersections. If there's overlap there as well, then we know that the ray intersects the box.

For each axis, we begin by computing the parameters where the ray (represented by the point P and vector $\mathbf{v}$) crosses the minimum and maximum planes. So for example, in the x direction we'll calculate intersections with the $x = x_{min}$ and $x = x_{max}$ planes. To do this, we need to solve the following equations:

$$P_x + s_x v_x = x_{min}$$
$$P_x + t_x v_x = x_{max}$$

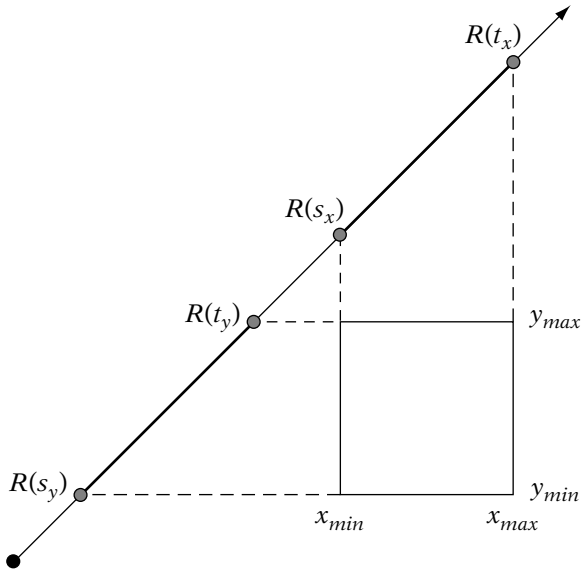


FIGURE 12.14 Axis-aligned box–ray separation test.

Solving for s_x and t_x, we get

$$s_x = \frac{x_{min} - P_x}{v_x}$$

$$t_x = \frac{x_{max} - P_x}{v_x}$$

To simplify adjustment of our overlap interval, we want to ensure that $s_x < t_x$. This can be handled by checking whether $1/v_x < 0$; if so, we'll swap the x_{min} and x_{max} terms.

We'll track our parameter overlap interval by using two values s_{max} and t_{min}, initialized to the maximum interval. For a ray this is $[0, \infty]$; for a line this would be $[-\infty, \infty]$; for a segment it would be $[0, s]$, where s is the length of the segment. These represent the maximum s and minimum t values seen so far. As we calculate intersection parameters for each axis, we'll sort them so that $s < t$, and then update s_{max} and t_{min} if $s > s_{max}$ or $t < t_{min}$. We know that the ray misses the box if we ever find that $s_{max} > t_{min}$. For example, looking at Figure 12.14, after doing the x-axis calculations we see that $s_{max} = s_x$ and $t_{min} = t_x$. After the y-axis parameters are computed, t_{min} is updated to t_y, and s_{max} remains s_x. But $s_x > t_y$, so there is no intersection.

The code, abbreviated for space, is as follows.

```
bool
IvAABB::Intersect( const IvRay3& ray )
{
    float maxS = 0.0f;   // for line, use -FLT_MAX
    float minT = FLT_MAX;   // for line segment, use length

    // do x coordinate test (yz planes)

    // compute sorted intersection parameters
    float s, t;
    float recipX = 1.0f/ray.mDirection.x;
    if ( recipX >= 0.0f )
    {
        s = (mMin.x - ray.mOrigin.x)*recipX;
        t = (mMax.x - ray.mOrigin.x)*recipX;
    }
    else
    {
        s = (mMax.x - ray.mOrigin.x)*recipX;
        t = (mMin.x - ray.mOrigin.x)*recipX;
    }

    // adjust min and max values
    if ( s > maxS )
        maxS = s;
    if ( t < minT )
        minT = t;
    // check for intersection failure
    if ( maxS > minT )
        return false;

    // do y and z coordinate tests (xz & xy planes)
    ...

    // done, have intersection
    return true;
}
```

There's one special case that is implicitly handled: Clearly, if v_x is zero, then there are no solutions for s_x and t_x; the ray is parallel to the minimum

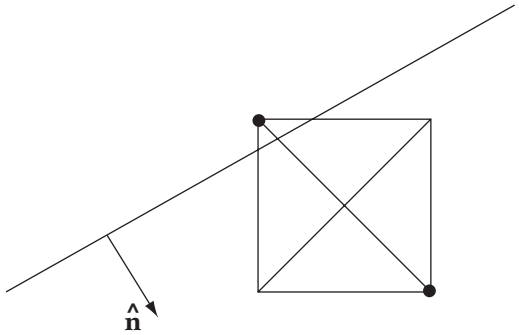


FIGURE 12.15 Axis-aligned box–plane separation test.

and maximum planes. Normally in this case, we'd need to test whether P_x lies between x_{min} and x_{max}. If not, the ray misses the box and there is no intersection. However, when using the IEEE floating-point standard, division by zero will return $-\infty$ for a negative numerator, and ∞ for a positive numerator. Hence, if the ray would miss the box, the resulting interval will be either $[-\infty, -\infty]$ or $[\infty, \infty]$, which will lead to intersection failure. More detail can be found in Williams et al. [119].

AABB–Plane Intersection

The most naive test to determine whether a box intersects a plane is to see whether a single box edge crosses the plane. That is, if two neighboring vertices lie on either side of the plane, there is an intersection. There are 12 edges, so this requires 24 plane tests. There are two improvements we can make to this. The first is to note that we need to test only *opposing* corners of the box; that is, two vertices that lie at either end of a diagonal that passes through the box center. This cuts the number of "edges" to be checked down to 4. The second improvement is provided by Möller and Haines [82], who note that we really need to test only one: the diagonal most closely aligned with the plane normal. Figure 12.15 shows a cross section of the situation.

Code to manage this is as follows. As before, we return zero if there is an intersection, the signed distance otherwise.

```
float
IvAABB::Classify( const IvPlane& plane )
{
    IvVector3 diagMin, diagMax;
    // set min/max values for x direction
    if ( plane.mNormal.x >= 0)
    {
```

```
            diagMin.x = mMin.x;
            diagMax.x = mMax.x;
        }
        else
        {
            diagMin.x = mMax.x;
            diagMax.x = mMin.x;
        }

        // ditto for y and z directions
        ...
        // minimum on positive side of plane, box on positive side
        float test = plane.mNormal.Dot( diagMin ) + plane.mD;
        if ( test > 0.0f )
            return test;

        test = plane.mNormal.Dot ( diagMax ) + plane.mD;
        // min on nonpositive side, max on nonnegative side, intersection
        if ( test >= 0.0f )
            return 0.0f;
        // max on negative side, box on negative side
        else
            return test;
    }
```

12.3.3 Swept Spheres

Definition

Source Code
Library
lvCollision
Filename
lvCapsule

The bounding sphere and the axis-aligned bounding box have one problem: There is no real sense of orientation. The sphere is symmetric across all axes and the AABB is always aligned to the world axes. For objects that have definite long and short axes (e.g., a human), this doesn't provide for an ideal approximation. The next two bounding objects we'll consider are not tied to the world axes at all, which makes them much more suitable for general models.

The simplest of such bounding regions are the swept spheres. If we consider the sphere as a region enclosed by a radius around a point, or a zero-dimensional center, the swept spheres use higher-dimensional centers. One example is the *capsule*, which is a line segment surrounded by a radius (Figure 12.16(a)). Another possibility is the lozenge, which has a quadrilateral center (Figure 12.16(b)). For our purposes, we'll concentrate on capsules (Eberly [25] provides more information on lozenges and other swept spheres.)

Computing the capsule in local space for a set of points is fairly straightforward, but not as simple as spheres or bounding boxes. Our first step is to

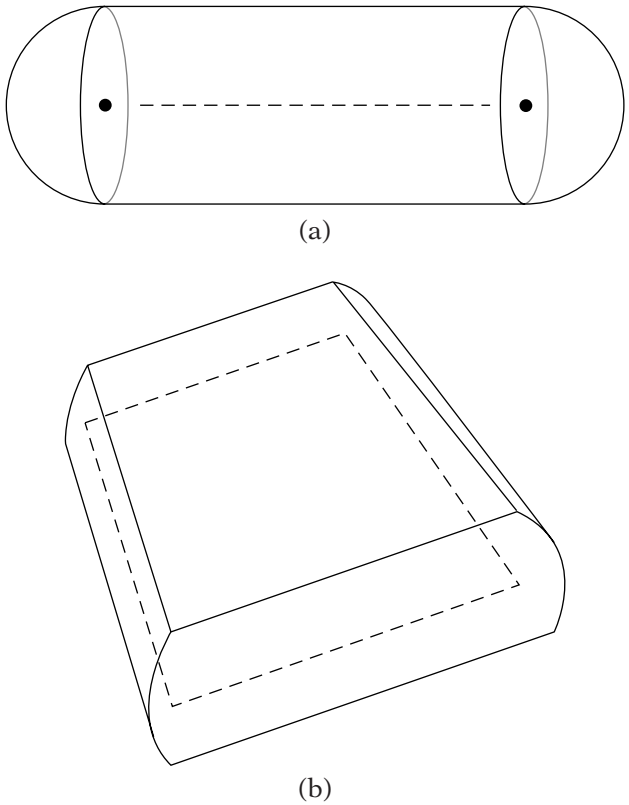


FIGURE 12.16 (a) Capsule and (b) lozenge.

compute a bounding box for the points. If the object is generally axis aligned (not unreasonable considering that the artists usually build objects in this way), we can use an axis-aligned bounding box. Otherwise we may need an oriented bounding box (see below on how to compute this). We then find the longest side. The line that we will use for our baseline segment runs through the middle of the box. We'll use the center of one end of the box as our line point A, and the box axis $\mathbf{w}$ as our line vector. We could use the local origin and a coordinate axis for our line, but while we're willing to assume axis alignment, we're not so optimistic as to assume that the object is centered on a coordinate axis.

Now we need to compute the radius r of the capsule. For each point in the object, we compute the distance from the point to the line. The maximum distance becomes our radius. The line combined with the radius gives us a tube with radius r and ends extending to infinity. All the points in the object just fit inside the tube.

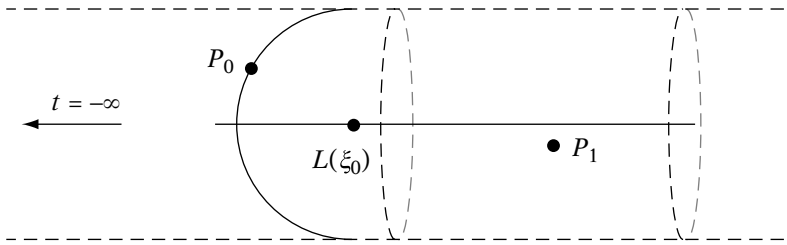


FIGURE 12.17 Capsule endcap fitting.

The final part to building the capsule is capping the tube with two hemispheres that just contain any points near the end of the object. Eberly [25] describes a method for doing this. The center of each hemisphere is one of the two endpoints of the line segment, so finding the hemisphere allows us to define the line segment. Let's consider the endpoint with the smaller t value — call it $L(\xi_0)$ — shown in Figure 12.17. We want to find the left-most hemisphere (i.e., the one with the smallest ξ_0) so that all points in the model either lie on the hemisphere (such as point P_0) or to the right of it (point P_1). Another way to think of this is that for each point we'll compute a hemisphere centered on the line that exactly contains that point and choose the hemisphere with the smallest ξ_0 value. If we do the same at the other end, with hemispheres oriented the other way and choosing the one with largest parameter value ξ_1, then all points will be tightly enclosed by the capsule.

To set this up, we first need to transform our points from the local space of the object to the local space of the line. We'll build a coordinate frame consisting of the line point A, normalized line vector $\hat{\mathbf{w}}$, and two vectors perpendicular to $\hat{\mathbf{w}}$: $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$. Subtracting the line point from the object point and multiplying by a 3×3 matrix formed from $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$, transforms the object-space point P to a line space point P' with line-space coordinates (u, v, w). Since $\hat{\mathbf{w}}$ is normalized, a point $L(\xi_0)$ on the line equals $(0, 0, \xi_0)$ in line space.

If P' lies on a hemisphere with radius r and center X_0 on the line, the length of a vector $\mathbf{d}$ from X_0 to P'_i should be equal to the radius r (Figure 12.18). Given this and the other parameters, we should be able to solve for X_0, and hence ξ_0.

The vector $\mathbf{d} = P' - X_0$. In line space, $\mathbf{d} = (u, v, w) - (0, 0, \xi) = (u, v, w - \xi)$. Ensuring that $\|\mathbf{d}\| = r$ means that

$$u^2 + v^2 + (w - \xi_0)^2 = r^2$$

Solving for ξ_0, we get

$$\xi_0 = w - \left(\pm\sqrt{r^2 - (u^2 + v^2)}\right)$$

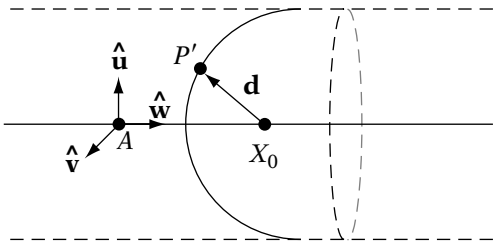


FIGURE 12.18 Determining hemisphere center X_0 for given point P'.

Since this is a hemisphere, we want X_0 to be to the right of P, so $w \geq \xi_0$, and this becomes

$$\xi_0 = w + \sqrt{r^2 - (u^2 + v^2)}$$

Computing this for every point P in our model and finding the minimum ξ_0 gives us our first endpoint. Similarly, the second endpoint is found by finding the maximum value of

$$\xi_1 = w - \sqrt{r^2 - (u^2 + v^2)}$$

Capsule–Capsule Intersection

Handling capsule–capsule intersection is very similar to sphere–sphere intersection. Instead of calculating the distance between two points, and determining whether that is less than the sum of the two radii, we calculate the distance between two line segments and check against the radii. As before, if the distance is less than the sum of the two radii, we have intersecting capsules.

```
bool
IvCapsule::Intersect( const IvCapsule& other )
{
    float radiusSum = mRadius + other.mRadius;
    return ( mSegment.DistanceSquared( other.mSegment )
                                          <= radiusSum*radiusSum );
}
```

Capsule–Ray Intersection

Capsule–ray intersection follows from capsule–capsule collision. Instead of finding the distance between two line segments, we need to find the distance between a ray and a line segment and compare them to the radius of the capsule, as follows.

```
bool
IvCapsule::Intersect( const IvRay3& ray )
{
    // test distance between line and segment vs. radius
    return ( ray.DistanceSquared( mSegment ) <= mRadius*mRadius );
}
```

Capsule–Plane Intersection

There are two tests necessary to determine whether a capsule intersects a plane. First of all, if the two endpoints of the line segment defining the capsule lie on either side of the plane, then clearly the capsule intersects the plane. However, even if the line segment lies on one side of the plane, the distance between one of the endpoints and the plane may be less than the radius. In this case, the capsule and plane would also intersect. Both cases are easy to test; we already have the pieces in place.

The code is as follows:

```
float
IvCapsule::Classify( const IvPlane& plane )
{
    float s0 = plane.Test( mSegment.GetEndpoint0() );
    float s1 = plane.Test( mSegment.GetEndpoint1() );

    // points on opposite sides or intersecting plane
    if (s0*s1 <= 0.0f)
        return 0.0f;

    // intersect if either endpoint is within radius distance of plane
    if( ::IvAbs(s0) <= mRadius || ::IvAbs(s1) <= mRadius )
        return 0.0f;

    // return signed distance
    return ( ::IvAbs(s0) < ::IvAbs(s1) ? s0 : s1 );
}
```

12.3.4 Object-Oriented Boxes

Definition

Source Code
Library
IvSimulation
Filename
IvOBB

World axis-aligned boxes are easy to create and fast to use for detecting intersections, but are not a very tight fit around objects that are not themselves generally aligned to the world axes (Figure 12.12). A more accurate approach is to create an initial bounding box that is a tight fit around the object in local space, and then rotate and translate the box as well as the object (Figure 12.19). These are known as object-oriented bounding boxes, or OBBs. This has another advantage in that we don't have to recalculate the box every time the object moves, just transform the initial one. Also, for rigid objects with a large number of vertices, recomputing the AABB every frame may be too expensive. The disadvantage is that testing intersections between two object-oriented boxes is more complicated. In the axis-aligned case, we could simplify our cases down to three tests because of the alignment. In the OBB case, the two can be at any relative orientation to each other, which complicates the issue considerably.

The representation for an OBB A consists of the center point C_a, an orientation matrix $\mathbf{R}_a$, and an extent vector $\mathbf{a}$ (Figure 12.20). The extent vector represents the difference from the center point to the point of maximum x, y, and z on the box. Note that the center of the box is not necessarily the same as the local origin of the object, nor does the orientation of the box have to match the orientation of the object. If either is the case, some adjusting of the object's local-to-world transformation will have to be done to generate the box axes and center location in world space. If the box to object-space orientation transformation is $\mathbf{R}_{box \to object}$ and the object's orientation is $\mathbf{R}_{object \to world}$, then the box's local-to-world rotation is

$$\mathbf{R}_{box \to world} = \mathbf{R}_{object \to world} \cdot \mathbf{R}_{box \to object}$$

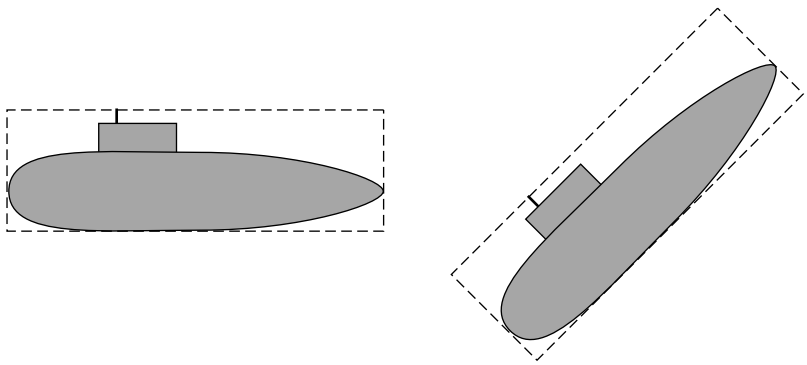


Figure 12.19 Object-oriented bounding boxes.

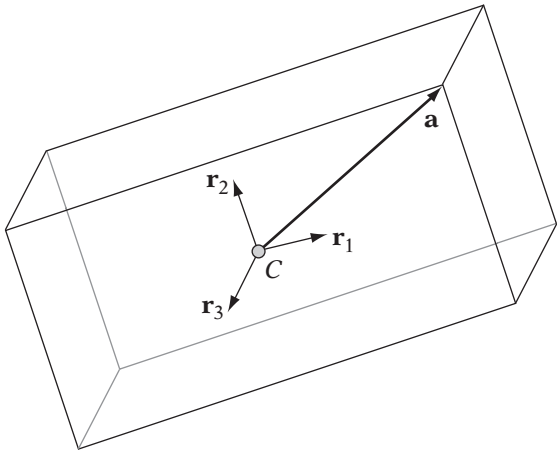


FIGURE 12.20 Properties of OBBs.

To simplify our life, however, we can use boxes aligned to the object's local coordinates, with a vector **d** in object space indicating the box center relative to the object center (as mentioned in Chapter 4, it's not usually practical to build objects with their bounding box center as their origin). In either case, any time we need the box center **c** in world space we can use

$$\mathbf{c} = \mathbf{R}_{object \to world}\,\mathbf{d} + \mathbf{t}$$

If we're simply simulating an object using an OBB, aligning it to the local axes may produce the results we want. However, when using an OBB for culling we often want a tighter fit than that. Ideally, we want to find the set of box axes that produce the minimum volume box, and there are a number of techniques that do just that. The most commonly used method approximates this by taking a statistical measure of the object known as the *covariance matrix* [113].

The covariance matrix is a 3×3 array represented as

$$\mathbf{C} = \begin{bmatrix} C_{xx} & C_{xy} & C_{xz} \\ C_{yx} & C_{yy} & C_{yz} \\ C_{zx} & C_{zy} & C_{zz} \end{bmatrix}$$

where C_{xy}, for example, is

$$C_{xy} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

The value $\bar{x}$ is the mean of x values of the n points, and $\bar{y}$ is the corresponding mean of the y values.

By computing the eigenvectors of this matrix, we can determine the direction of greatest variance or where the points are most spread out, which will become our long axis. The other eigenvectors become the directions of the remaining axes for our OBB. The mean of the points becomes the center of the box, and from there we can project the points onto the axes to determine the maximum extent along each axis. Code for this computation can be found on the accompanying CD-ROM.

OBB–OBB Intersection

There have been many methods for testing intersections between two arbitrarily oriented boxes, including linear programming techniques and closest-feature tracking. The most efficient technique known to date, however, uses the concept of separating axes and is due to Gottschalk et al. [46]. The following discussion is heavily drawn from this paper, with some additional concepts due to Eberly [25] and van den Bergen [112].

Recall that to test whether two axis-aligned boxes were intersecting, we did three tests, one for each axis x, y, and z. For each test, we checked the extents of each box along each of the axis directions. This is equivalent to projecting the box along the basis vectors **i**, **j**, and **k**. If the intervals of a given projection don't overlap, then there is a separating plane normal to the test vector and therefore no intersection. The corresponding axis is known as a *separating axis*.

This works well for axis-aligned boxes, but we need a slightly different test for oriented boxes. To simplify our equations and improve performance, we'll use transformations relative to box A. We end up with a single translation vector $\mathbf{c}$ from A to B, where $\mathbf{c} = \mathbf{R}_a^T \cdot (C_b - C_a)$, and a relative rotation matrix $\mathbf{R} = \mathbf{R}_b^T \mathbf{R}_a$. A's extent vector remains the same, since it's relative to its local space. B's extent vector becomes $\mathbf{R}^T \mathbf{b}$.

Now suppose we have a potential separating axis direction $\mathbf{v}$. We want to perform the same test we did with the AABBs: project each box onto the vector and check to see whether the projections are separate or not. Another way of representing this is to project the box centers onto the vector as endpoints, and then project the extent vectors closest to the center onto the vector as well (Figure 12.21). If the distance between the projected box centers is less than the sum of the lengths of the projected extents, then there is no intersection. Expressed mathematically, there is no intersection if

$$|\mathbf{c} \cdot \mathbf{v}| > |\mathbf{a} \cdot \mathbf{v} + (\mathbf{R}^T \mathbf{b}) \cdot \mathbf{v}|$$

This works if the extent vectors are aligned appropriately to give us the maximum projected length, but we can't make that assumption. Instead, we'll

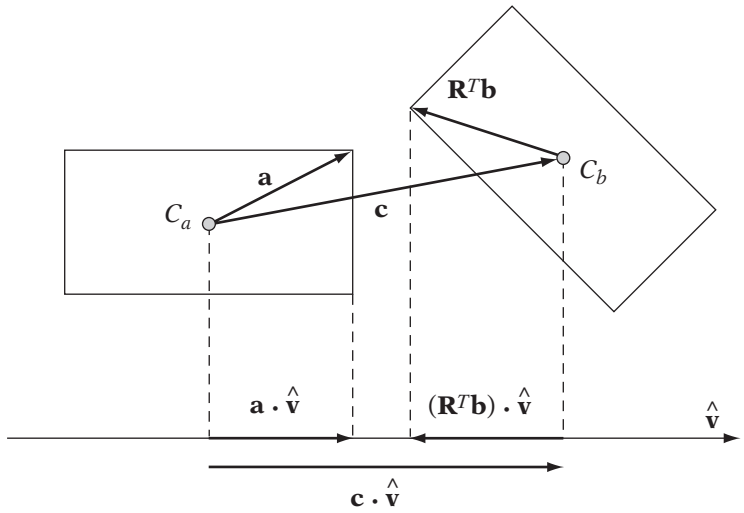


FIGURE 12.21 Example of OBB separation test.

use a pseudo–dot product that forces maximum length, so the equivalent to $\mathbf{a} \cdot \mathbf{v}$ is

$$|a_x v_x| + |a_y v_y| + |a_z v_z|$$

This is legal because the extents can be taken from any of the eight octants, so we can get any sign we want for any term.

An equivalent equation can be found for $(\mathbf{R}^T\mathbf{b}) \cdot \mathbf{v}$. The final separating axis equation is

$$|\mathbf{c} \cdot \mathbf{v}| > \sum_i |a_i v_i| + \sum_i |(\mathbf{R}^T\mathbf{b})_i v_i| \tag{12.11}$$

While this gives us our test, there is an infinite number of choices for $\mathbf{v}$, which is not practical. Gottschalk et al. [46] demonstrates that any separating plane either will be parallel to one of the box faces or parallel to an edge from each box. This means that a maximum of 15 separating axis tests are necessary: 3 against the axes of box A, 3 against the axes of box B, and 9 cross products using one axis from A and one from B.

The nice thing about this result is that it allows us to simplify our equations considerably. For example, let's use the cross product of the local x-axis from A and the local y-axis from B. In A's local space, the x-axis of A is $\mathbf{i} = (1, 0, 0)$. If we represent the matrix $\mathbf{R}$ as the three row vectors $(\mathbf{r}_0^T, \mathbf{r}_1^T, \mathbf{r}_2^T)$, then the y-axis of B in A's space is (r_{10}, r_{11}, r_{12}). Performing the cross product $\mathbf{i} \times \mathbf{r}_1$, we get

$$\mathbf{v} = (0, -r_{12}, r_{11}) \tag{12.12}$$

For the B terms, it's convenient to transform $\mathbf{v}$ to be relative to B's basis via $\mathbf{R}^T$:

$$\mathbf{R}^T(\mathbf{i} \times \mathbf{r}_1) = \begin{bmatrix} \mathbf{r}_0 \cdot (\mathbf{i} \times \mathbf{r}_1) \\ \mathbf{r}_1 \cdot (\mathbf{i} \times \mathbf{r}_1) \\ \mathbf{r}_2 \cdot (\mathbf{i} \times \mathbf{r}_1) \end{bmatrix} = \begin{bmatrix} \mathbf{i} \cdot (\mathbf{r}_1 \times \mathbf{r}_0) \\ \mathbf{i} \cdot (\mathbf{r}_1 \times \mathbf{r}_1) \\ \mathbf{i} \cdot (\mathbf{r}_1 \times \mathbf{r}_2) \end{bmatrix} = \begin{bmatrix} \mathbf{i} \cdot (-\mathbf{r}_2) \\ \mathbf{i} \cdot \mathbf{0} \\ \mathbf{i} \cdot \mathbf{r}_0 \end{bmatrix}$$

So, $\mathbf{v}$ in B space is

$$\mathbf{R}^T \mathbf{v} = (-r_{20}, 0, r_{00}) \tag{12.13}$$

Substituting equations 12.12 and 12.13 into equation 12.11 and multiplying out the terms, the final axis test is

$$|c_2 r_{11} - c_1 r_{12}| > a_1 |r_{12}| + a_2 |r_{11}| + b_0 |r_{20}| + b_2 |r_{00}|$$

The test for other axes can be derived similarly. All use the absolute value of elements from the matrix $\mathbf{R}$ so it is far more efficient to precompute them and then perform the axis tests. If this is done, the algorithm takes about 200 operations. It can be found in `IvOBB::Intersect()`.

One caveat: Any implementation of this algorithm needs to take steps to avoid numerical problems with floating-point precision. In particular, if two edges, one from each box, are nearly parallel, the resulting cross product will be near zero. This will lead to invalid results for the separation test. The solution is to detect the condition, and only test against the six main axes of the boxes. Even in this case, care must be taken as numeric error can lead to false negatives.

OBB–Ray Intersection

Detecting intersection between a linear component and an oriented box is much simpler than detecting intersection between two boxes. One method is to transform the ray into the box's local space and perform a standard AABB intersection test. To transform the linear component, the origin point is transformed by the inverse of the box's world transform matrix, and the direction vector by the inverse rotation of the box's transformation matrix. The newly transformed line, ray, or line segment can be passed into the appropriate AABB routine.

An alternative is to use a modified version of the AABB algorithm, as described by Möller and Haines [82]. In this case, instead of using planes normal to the three world axes, we'll use planes normal to the three box axes. Recall that these axes are specified as the three column vectors in our rotation matrix.

Each axis has two parallel planes associated with it. If we treat the box's center as the origin of our frame, the extent vector **a** contains the magnitude of our d values for these planes. For example, two of the parallel box planes are $r_{00}x + r_{10}y + r_{20}z + a_x = 0$ and $r_{00}x + r_{10}y + r_{20}z - a_x = 0$.

If we translate our ray so that its origin is relative to the box origin, we can determine s and t parameters for the intersections with these planes, just as we did with the axis-aligned box. In this case, the formulas for s and t for each axis (including the translation) are

$$s = \frac{\mathbf{r}_i \cdot (C - P) - \mathbf{a}_i}{\mathbf{r}_i \cdot \mathbf{v}} \qquad t = \frac{\mathbf{r}_i \cdot (C - P) + \mathbf{a}_i}{\mathbf{r}_i \cdot \mathbf{v}}$$

We also need to modify our test to determine whether the ray is parallel to the current pair of planes we're testing. This is easily done by taking the dot product of the direction vector **v** and the plane normal and seeing if it is close to zero. If so, the ray is parallel to the plane, and we need to project the vector $C - P$ onto the current axis, and see if the result lies outside the extents.

The modified code is as follows:

```
bool
IvOBB::Intersect( const IvRay3& ray )
{
    float maxS = -FLT_MAX;
    float minT = FLT_MAX;

    // compute difference vector
    IvVector3 diff = mCenter - ray.mOrigin;

    // for each axis do
    for (int i = 0; i < 3; ++i)
    {
        // get axis i
        IvVector3 axis = mRotation.GetColumn( i );
        // project relative vector onto axis
        float e = axis.Dot( diff );
        float f = ray.mDirection.Dot( axis );

        // ray is parallel to plane
        if ( ::IsZero( f ) )
        {
            // ray passes by box
            if ( -e - mA[i] > 0 || -e + mA[i] > 0 )
               return false;
            continue;
```

```
            }

            float s = (e - mA[i])/f;
            float t = (e + mA[i])/f;

            // fix order
            ...
            // adjust min and max values
            ...
            // check for intersection failure
            ...
        }

        // done, have intersection
        return true;
    }
```

Performance can be improved here by storing the rotation matrix as an array of three vectors instead of an `IvMatrix33`.

OBB–Plane Intersection

As we did with with OBB–ray intersection, we can classify the intersection between an OBB and a plane by transforming the plane to the OBB's frame and using the AABB–plane classification algorithm. Since the transformation is just a pure rotation and a translation, we can find the transformed normal by

$$\hat{\mathbf{n}}' = \mathbf{R}^T \hat{\mathbf{n}}$$

We apply the transpose since we're going from world space into box space. The minimal and maximal points for the AABB in this case are the extent vector and its negative, $\mathbf{a}$ and $-\mathbf{a}$, respectively.

An alternative, presented by Möller and Haines [82], is to use the principle of separating planes again. This time, our test vector will be the plane normal, and we'll project the box diagonal on to it. To ensure we get maximum extent, we'll add the absolute values of the elements together, similar to what we did before:

$$r = |(a_0 \mathbf{r}_0) \cdot \mathbf{n}| + |(a_1 \mathbf{r}_1) \cdot \mathbf{n}| + |(a_2 \mathbf{r}_2) \cdot \mathbf{n}|$$

Here, each $\mathbf{r}_i$ represents a column of the rotation matrix. The box intersects the plane if the distance between the box center and the plane is less than r.

The resulting code is as follows:

```
float IvOBB::Classify( const IvPlane& plane )
{
    IvVector3 xNormal = ::Transpose(mRotation)*plane.mNormal;
    float r = mExtents.x*::IvAbs(xNormal.x) + mExtents.y*::IvAbs(xNormal.y)
        + mEextents.z*::IvAbs(xNormal.z);

    float d = plane. Test(mCenter);
    if (::IvAbs(d) < r)
       return 0.0f;
    else if (d < 0.0f)
       return d + r;
    else
       return d - r;
}
```

12.3.5 Triangles

Source Code
Library
IvMath
Filename
IvTriangle

All of the bounding objects we've discussed up until now have been approximations to our base object (assuming our object is more complex than, say, a box or a sphere). To test actual intersections between objects, we need to get right down to the basic building block of our geometry: the triangle. As before, we will be representing our triangle as the convex combination of three points.

Triangle–Triangle Intersection

A naive approach to determining triangle–triangle intersection uses the triangle–ray intersection test that follows. If one of the line segments composing an edge of one triangle intersects the other triangle, then the two triangles are intersecting. While this works, there are faster methods. Two commonly used approaches are by Möller [80] and Held [58]. However, if we are only concerned with determining whether intersection exists, and not the segment (or point) of intersection, then there is a faster way, concurrently discovered by two groups of researchers: Shen et al. [102] and Guigue and Devillers [51].

Figure 12.22 shows the situation. Taking the first triangle P, composed of points P_0, P_1, and P_2, we compute its plane equation. Recall that the plane equation for a normal $\mathbf{n} = (a, b, c)$ and a point on the plane $P_0 = (x_0, y_0, z_0)$ is

$$0 = ax + by + cz - (ax_0 + by_0 + cz_0)$$

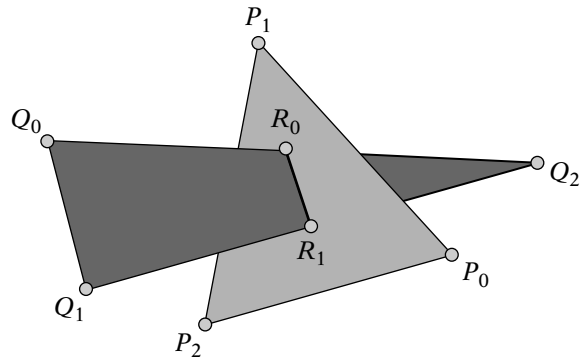


FIGURE 12.22 Triangle intersection.

or

$$0 = ax + by + cz + d$$

In this case, the plane normal is computed from $(P_1 - P_0) \times (P_2 - P_0)$ and normalized, and the plane point is P_0.

Now we take our second triangle Q, composed of points Q_0, Q_1, and Q_2. We plug each point into P's plane equation and test whether all three lie on the same side of the plane. This is true if all three results have the same sign. If they do, there is no intersection and we quit. Otherwise, we store the results d_0, d_1, and d_2 generated from the plane equation for each point and continue.

We now need to test whether the rearranged triangles overlap by checking the intervals where their edges cross the common line between the two planes. If the interval for P is $[i, j]$ and Q is $[k, l]$, then there is intersection if the intervals overlap, producing the line segment $[R_0, R_1]$. Other algorithms compute these intervals directly. However, there is a way to test this implicitly.

First, we rearrange P's vertices such that the lone vertex (the one that lies in its own half-space of Q) is first, or P_0. We also permute Q's vertices so that P_0 will "see" them in counterclockwise order. We then do the same for triangle Q, rearranging its vertices such that its lone vertex is first, and permuting P's vertices into counterclockwise order relative to the new Q_0.

Now, we make use of a signed distance test to check for interval overlap. If the signed distance between Q_0Q_1 and P_0P_2 is negative, then there is no overlap. Similarly, if the signed distance between P_0P_1 and Q_0Q_2 is negative, there is no overlap. Otherwise, the two triangles intersect.

We compute the signed distance between two edges by comparing the distance between two parallel planes, each containing one of the line segments. The normal **n** for these planes can be computed by taking the cross product

between the segment vectors, say $\mathbf{n} = (Q_0 - Q_1) \times (P_0 - P_2)$. Then, we can compute the signed distance between each plane and the origin by taking the dot product of the plane normal with a point on each plane (i.e., $d_0 = \mathbf{n} \cdot Q_0$ and $d_1 = \mathbf{n} \cdot P_0$). Then, the signed distance between the planes is just $d_0 - d_1$, or $\mathbf{n} \cdot (Q_0 - P_0)$.

Note that this will not work if the two lines are parallel. Most of the cases where this might occur are culled out during the initial steps. The one case remaining is if the two triangles are coplanar. This is handled by projecting them to 2D and doing a simple test.

Triangle–Ray Intersection

There are two possible approaches to determining triangle–ray intersection. The first is to use the plane equation for the triangle (computed from the three vertices) and determine the intersection point of the ray with the plane (if any). We can then use a point-in-triangle test to determine whether the intersection lies within the triangle.

While a relatively simple approach, it has some disadvantages. First of all, we need to either store the plane equation or, if we're short on space, compute it every time we wish to do the intersection test. Second, it's a two-pass algorithm: compute the plane intersection, and then test whether it's in the triangle. Fortunately, we have an alternative. The following approach, presented by Möller and Trumbore [81], uses affine combinations to compute the ray–triangle intersection.

We define our triangle as having vertices V_0, V_1, and V_2. We can define two edge vectors $\mathbf{e}_0$ and $\mathbf{e}_1$ (Figure 12.23), where

$$\mathbf{e}_0 = V_1 - V_0$$
$$\mathbf{e}_1 = V_2 - V_0$$

Recall that the point V_0 with the vectors $\mathbf{e}_0$ and $\mathbf{e}_1$ can be used to create an affine combination that spans the plane of the triangle, with barycentric coordinates (u, v). So, the formula for a point $T(u, v)$ on the plane is

$$\begin{aligned} T(u, v) &= V_0 + u\,\mathbf{e}_0 + v\,\mathbf{e}_1 \\ &= V_0 + u(V_1 - V_0) + v(V_2 - V_0) \end{aligned}$$

Rearranging terms, we get

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

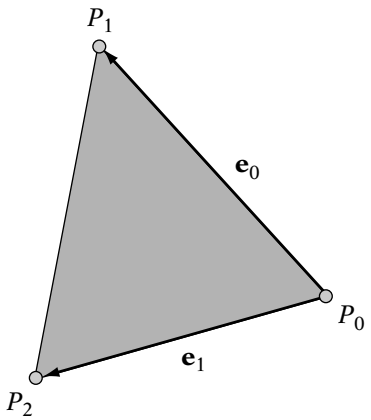


FIGURE 12.23 Affine space of triangle.

We want the contribution of each point to be nonnegative, so for a point inside the triangle,

$$u \geq 0$$
$$v \geq 0$$
$$u + v \leq 1$$

If u or $v < 0$, then the point is on the outside of one of the two axis edges. If $u + v > 1$, the point is outside the third edge. So, if we can compute the barycentric coordinates for the intersection point $T(u, v)$, we can easily determine whether the point is outside the triangle.

To compute the u, v coordinates of the intersection point, the result of the line equation $L = P + t\,\mathbf{d}$ will equal a solution to the affine combination $T(u, v)$ (Figure 12.24). So,

$$P + t\,\mathbf{d} = (1 - u - v)V_0 + uV_1 + vV_2$$

We can express this as a matrix product:

$$\begin{bmatrix} -\mathbf{d} & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = P - V_0$$

Using Cramer's rule, or row reduction, we can solve this matrix equation for (t, u, v). The final result is

$$t = \frac{\mathbf{q} \cdot \mathbf{e}_2}{\mathbf{p} \cdot \mathbf{e}_1}$$

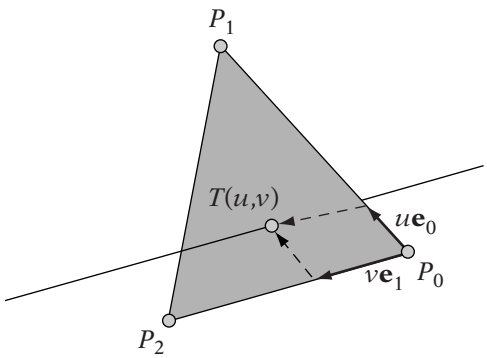


FIGURE 12.24 Barycentric coordinates of line intersection.

$$u = \frac{\mathbf{p} \cdot \mathbf{s}}{\mathbf{p} \cdot \mathbf{e}_1}$$

$$v = \frac{\mathbf{q} \cdot \mathbf{d}}{\mathbf{p} \cdot \mathbf{e}_1}$$

where

$$\mathbf{e}_1 = V_1 - V_0$$

$$\mathbf{e}_2 = V_2 - V_0$$

$$\mathbf{s} = P - V_0$$

$$\mathbf{p} = \mathbf{d} \times \mathbf{e}_2$$

$$\mathbf{q} = \mathbf{s} \times \mathbf{e}_1$$

The final algorithm includes checks for division by zero and intersections that lie outside the triangle.

```
bool
TriangleIntersect( const IvVector3& v0, const IvVector3& v1,
                     const IvVector3& v2, const IvRay& ray )
{
  // test ray direction against triangle
  IvVector3 e1 = v1 - v0;
  IvVector3 e2 = v2 - v0;
  IvVector3 p =  ray.mDirection.Cross(e2);
  float a = e1.Dot(p)
```

```
    // if result zero, no intersection or infinite intersections
    // (ray parallel to triangle plane)
    if ( ::IsZero(a) )
      return false;

    // compute denominator
    float f = 1.0f/a;

    // compute barycentric coordinates
    IvVector3 s = ray.mOrigin - v0;
    u = f*s.Dot(p)
    if (u < 0.0f || u > 1.0f) return false;

    IvVector3 q = s.Cross(e1);
    v = f*ray.mDirection.Dot(q);
    if (v < 0.0f || u+v > 1.0f) return false;

    // compute line parameter
    t = f*e2.Dot(q);

    return (t >= 0);
  }
```

Parameters u, v, and t can be returned if the barycentric coordinates on the triangle or the parameter for the exact point of intersection are needed.

Triangle–Plane Intersection

We covered triangle–plane intersection when we discussed triangle–triangle intersection. We take our triangle, composed of points P_0, P_1, and P_2, and plug each point into the plane equation. If all three lie on the same side of the plane, then there is no intersection. Otherwise, there is, and if we desire we can find the particular line segment of intersection, as described earlier. If there is no intersection, the signed distance is the plane equation result of minimum magnitude.

12.4 A Simple Collision System

Now that we have some methods for testing intersection between various primitive types, we can make use of them in a practical system. The example we'll consider is collision detection. Rather than building a fully general collision system, we'll do only as much as we need to for a basic game — in our

case, we'll use a submarine game as our example. This is to keep things as simple as possible and to illustrate various points to consider when building your own system. It's also good to keep in mind that a particular subsystem of a game, whether it is collision or rendering, needs only to be as accurate as the game calls for. Building a truly flexible collision system that handles all possible situations may be overkill and eat up processing time that could be used to do work elsewhere.

12.4.1 Choosing a Base Primitive

The first step in building the system is to choose the base bounding shape for our objects. We'll see in the following sections how we can use a hierarchy of bounding primitives to get a better fit to the object's surface, but for now we'll consider only one per object. Which primitive we choose depends highly on the expected topology we're trying to approximate with it. For example, if we're writing a pool game, using bounding spheres for our balls makes perfect sense. However, for a human character bounding spheres are not a good choice because one axis of the object is far longer than the other two — not a good fit. In particular, getting characters through an interior space might be a tricky proposition unless all your doorways and hallways are at least six feet wide.

Considering that our object is made of triangles, using them should give us the most accurate results. However, while they are cheap as a one-on-one test, it would be costly to test every possible triangle–triangle combination between two objects. This becomes more feasible when we have some sort of culling hierarchy to whittle down the possible triangle pairs to a few contenders — we'll discuss that in more detail shortly. However, if we can get a good fit with a simpler bounding volume, we can get a reasonably accurate measure of collision by doing a volume–volume test without having to do the full triangle–triangle test.

Since AABBs change size depending on the object's orientation, they are not usually a good choice for a base bounding primitive. They are more often used as a culling test, such as in the sweep-and-prune system described in Section 12.4.4.

Among the primitives we've discussed, this leaves us with capsules and OBBs. Which we choose depends on our performance requirements and how angular our objects are. If we have mostly boxy objects — like tanks — capsules or even lozenges won't provide very compelling collisions. An OBB is a better shape to choose for this situation. For our case, however, submarines and torpedoes are both generally sausage shaped. If we had to go with a single bounding object that approximates a submarine, capsules are an excellent choice.

12.4.2 Bounding Hierarchies

Source Code Demo Hierarchy

Unless our objects are almost exactly the shape of the bounding primitive (such as our pool ball example), then there are still going to be places where our test indicates intersection where there is visibly no collision. For example, the conning tower of our submarine makes the bounding capsule encompass a large area of empty space at the top of the hull. Suppose a torpedo is heading toward our submarine and through that area. Instead of harmlessly passing over the hull as we would expect from the visual evidence, it will explode because we have detected a collision with the inaccurately large bounding region.

The solution is to use a set of bounding primitives to get a better approximation to the surface of the object. In our submarine example, we could use one capsule for the main hull and one for the conning tower. If we are willing to allow a slightly forgiving system, we could ignore the conning tower for the purposes of collision and get a very nice fit with the hull capsule. Or we could go the more detailed route and add one for the conning tower, as well as a third for the periscope (Figure 12.25). To check for intersection, we test each bounding primitive for the first object against all the primitives in the second, much as we would have done for the triangles.

To speed this up we can keep our original bounding capsule and use it as a rough test before checking further. Better still, we can generate bounding spheres for each object and test against those instead. It's a very cheap test and can do a great job of culling large numbers of cases. We could also generate bounding spheres for each of our smaller capsules and use these spheres in preliminary culling steps before checking individual capsule pairs.

This gives us a bounding hierarchy for our object (Figure 12.26). We compare the top-level bounding spheres first. Only if they are intersecting do we then move on to the lower level of sphere check and capsule check. This can cull out a large number of cases and make it much more likely that we'll be testing only the two lower-level capsules that are actually intersecting.

We can take this technique of using bounding hierarchies further. For example, if we want to do triangle–triangle intersection testing, we can build

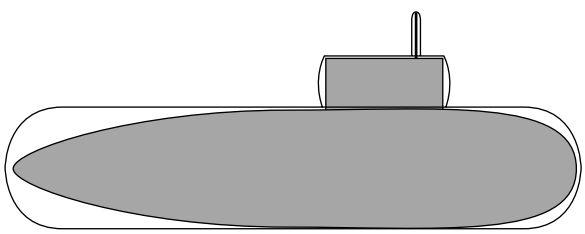


Figure 12.25 Using multiple bounding objects.

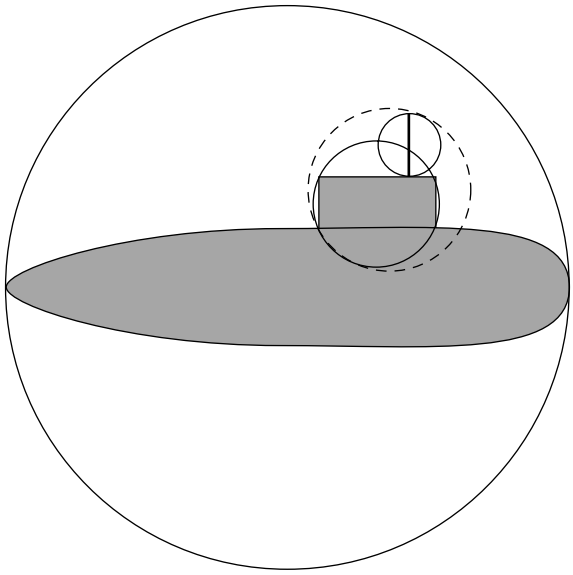


Figure 12.26 Using bounding hierarchy.

a hierarchy to perform coarser but cheaper intersection tests. If two objects are intersecting, we can traverse the two hierarchies until we get to the two intersecting triangles (there may be more than two if the objects are concave). Obviously, we'll want to create much larger hierarchies in this case. Generating them so that they are as efficient as possible — they both cull well and have a reasonably small tree size — is not a simple task. Gottschalk et al. [46] provide some information for building OBB trees, while Ericson [32] covers the general cases.

Spheres, capsules, AABBs, and OBBs have all been used as primitives for culling bounding hierarchies. Most tests have been done for hierarchies with triangles as leaf nodes. Gottschalk et al. [46] demonstrate that OBBs work better than both AABBs and spheres if our objects have static geometry. However, if we're constantly deforming our vertices — for example, with skinned character models — recomputing the OBBs in the hierarchy is an expensive step. Using spheres or AABBs can be a better choice in this circumstance.

12.4.3 Dynamic Objects

So far we have been using intersection tests assuming that our objects don't move between frames. This is clearly not so. In games, objects are constantly moving, and we need to be careful when we use static tests to catch collisions between moving objects.

For example, in one frame we have two objects moving toward each other, clearly heading for a collision somewhere in the center of the screen (Figure 12.27(a)). Ideally, in the next frame we want to catch a snapshot of them just as they collide or are slightly intersecting. However, if we take too large a simulation step, they may have passed partially through each other (Figure 12.27(b)). Using a frame-by-frame static test, we will miss the initial collision. Worse yet, if we take a larger step, the two objects will have passed right through each other, and we'll miss the collision entirely.

One way to catch this is to sweep our bounding primitives along a path and then test intersection between the swept primitives that we've generated. A simple example of this is testing intersection between two moving spheres. If we sweep a sphere along a line segment, we get — no surprise — a capsule. Based on the two objects' velocities, we can generate capsules for each object and test for intersection. If one is found, then we know the two objects may collide somewhere between frames and we can investigate further.

We generally have to worry about this problem only when the relative velocities of objects are large enough or the frame times are long enough that one object can move, relative to another, farther than half its thickness in the direction of travel. For example, a tank with a speed of 30 km/hr moves about 0.12 m/frame, assuming 60 frames/s. If the tank is 10 meters long, its movement is miniscule compared to its total length and we can probably get away with static testing. Suppose, however, that we fire a 1 meter–long missile

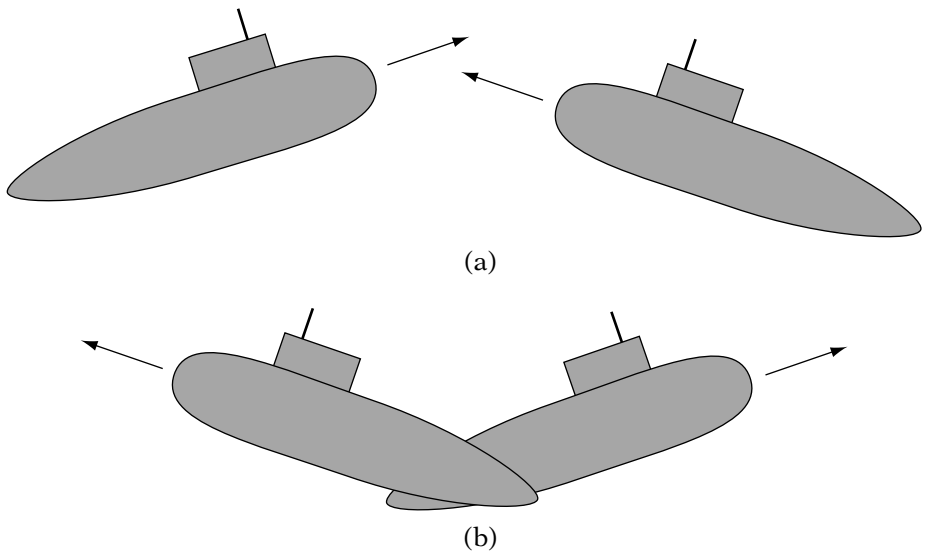


FIGURE 12.27 (a) Potential collision, and (b) partially missed collision.

at that tank, traveling at 120 km/hr. We also have a bug in our rendering code that causes us to drop to 10 frames/s, giving us a travel distance of 3 1/3 meters. The missile's path crosses through the tank at an angle and is already through it by the next frame. This may seem like an extreme example, but in collision systems it's often best to plan for the extreme case.

Walls, since they are infinitely thin, also insist on a dynamic test of some kind. In a first-person shooter you don't want your players using a cheat to teleport through a wall by moving too fast. One way to handle this is to do a simple test of the player's path versus the nearest wall plane. Another is to create a plane for each wall with the normals pointing into the room; if a plane test shows that the object is on the negative side of the plane, then it's no longer in the room.

Submarines are large and move relatively slowly for their size, so for this collision system we don't need to worry about this issue. However, it is good to be aware of it. For more information on managing dynamic tests, see Eberly [25].

12.4.4 Performance Improvements

Now that we've handled questions of which bounding shapes to use on our objects and how to achieve a tighter fit even with simple primitives, we'll consider ways of improving our performance. The main way we'll approach this is to cut down on intersection tests. We've already handled this to some extent at the object level by using a bounding hierarchy to cut down on intersection tests between primitives. Now we want to look at the world level, by cutting down on tests between objects. For example, if two objects are relatively small and at opposite ends of the map from each other, it's a pretty good bet that they're not colliding.

The most basic way to check collisions among all objects is the following loop:

```
for each object i
    for each object j, where j <> i
         test for collision between i and j
```

There are a number of problems with this. First of all, we're doing $n(n-1)$ tests, which is an $O(n^2)$ algorithm. Half of those tests are duplicates: If we test for collision between objects 1 and 5, we'll also test for collision between 5 and 1. Also, there may be a number of objects that we wish to collide with that simply aren't moving. We don't want to test collision between two such static objects. A better loop that handles these cases is as follows:

```
for each object i
    for each object j, where j > i
         if (i is moving or j is moving)
             test for collision between i and j
```

There are other possibilities. We can have two lists: one of moving objects called `Colliders` and one of moving or static objects called `Collidables`. In the first loop we iterate through the `Colliders` and in the second the `Collidables`. Each `Collider` should be tagged after its turn through the loop, to ensure collision pairs aren't checked twice. Still, even with this change, we're still doing $O(nm)$ tests, where n is the number of `Colliders` and m is the number of `Collidables`. We need to find a way to further cut down the number of checks.

Most approaches involve some sort of spatial subdivision to do this. The simplest is to slice the world, along the x-axis say, by a series of evenly spaced planes (Figure 12.28). This creates a set of slabs, bounded by the planes along the x direction, and by whatever bounds we've set for our world in the y and

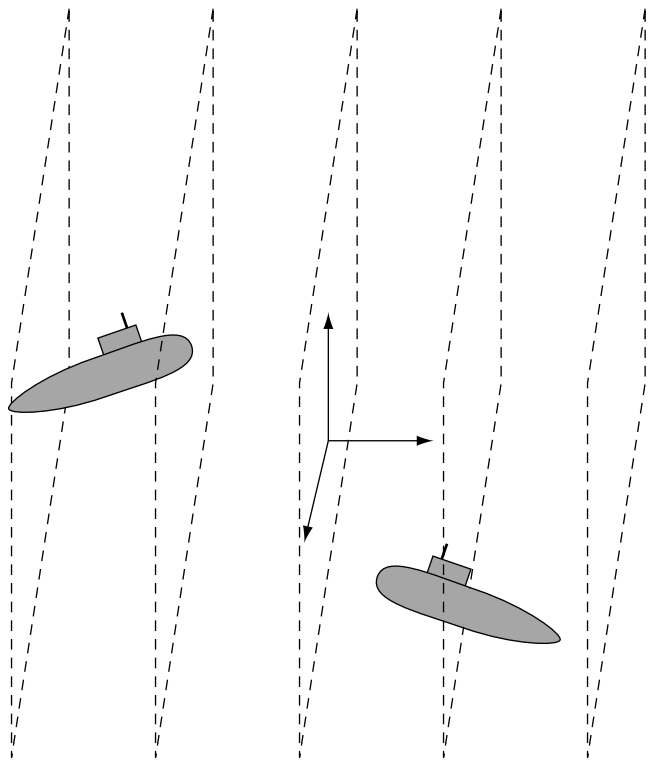


Figure 12.28 Cutting collision space into slabs.

z directions. For each slab, we store the set of objects that intersect it. To test for collisions for a particular object, we determine which slabs it intersects and then test against only the objects in those slabs. This approach can be extended to other spatial subdivisions, such as a grid or voxel-based system.

One of the disadvantages of the regular spatial subdivisions is that they don't handle clumping very well. Let's consider slabs again. If our world is fairly sparse, there may be large numbers of slabs with no objects in them, and a very few with most of the objects in them. We still may end up doing a large number of checks within each slab, which is the problem we were trying to avoid.

There is another possibility used by a number of collision-detection systems, known as the *sweep-and-prune* method. It is similar to the separating axis test that we used for OBBs (it's also related to some scan line rasterization algorithms). Instead of using a regular grid for our world, we'll use the extents of our objects as our grid. For each object, we project its extents onto the x-axis. To keep things efficient, we can use our root-level bounding sphere to compute our extents, which for a sphere with center C and radius r, gives us an interval of $[c_x - r, c_x + r]$.

Given the extent endpoint pairs for each object, we'll mark them with a pointer to the object and indicate for each value whether it is the low (*start*) or high (*finish*) endpoint. Finally, we sort all endpoints from low to high.

Once the sorted list of endpoints is created, the collision-detection process runs as follows.

```
for each endpoint do
   if a start point
   if object is moving
      check collisions against all objects in list
   else
      check collisions against moving objects in list
   add corresponding object to list
else if a finish point
   remove corresponding object from list
```

Figure 12.29 shows how this works. We sweep from left to right along the x-axis and use the sorted endpoints to test intersections of intervals before the more complex intersection tests.

Normally this would be an $\Omega(n \log n)$ algorithm due to the sorting operation. However, if the time step is small enough, the relative position of the objects won't have changed that much from frame to frame — this is referred to as *temporal coherence*. Any changes that do happen will be rare but localized. Therefore, if we use a sorting algorithm that works best on mostly sorted

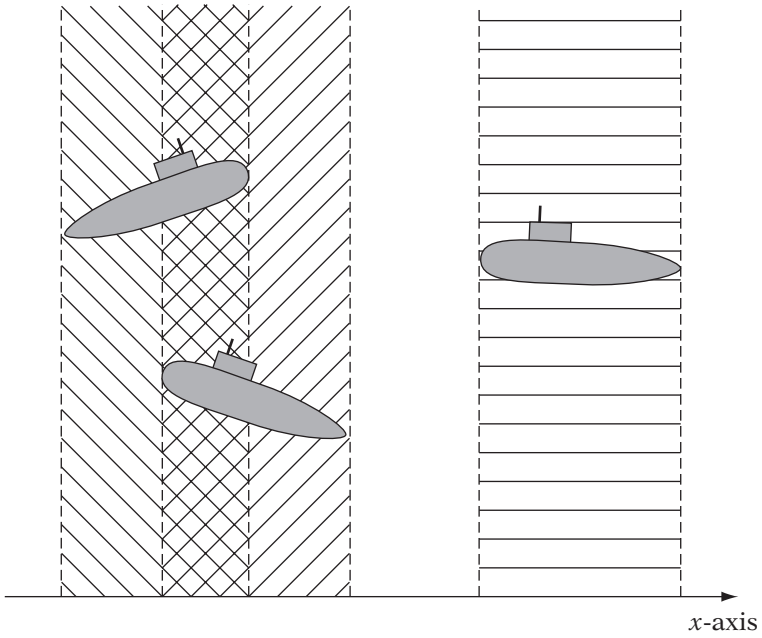


FIGURE 12.29 Dividing collision space by sweep and prune.

lists, such as bubble or insertion sort, we can get linear time for our sort and hence an $O(n)$ algorithm.

This algorithm still has problems, of course. If our objects are highly localized (or clumped) in the x direction, but separated in the y direction, then we still may be doing a high number of unnecessary intersection tests. But it is still much better than the naive $O(n^2)$ algorithm we were using before.

12.4.5 Related Systems

The other two systems we mentioned earlier were ray casting, for picking and AI tests, and frustum culling. Both systems can benefit from the techniques described in our collision system, in particular the use of bounding hierarchies and spatial partitioning.

Consider the case of ray casting. Instead of testing the ray directly against the object, we can take the ray and pass it through the hierarchy until (if we desire) we get the exact triangle of intersection. Further culling of testing can be done by using a spatial partitioning system such as voxels or *kd* trees to consider only those objects that lie in the areas of the spatial partitioning that intersect the ray.

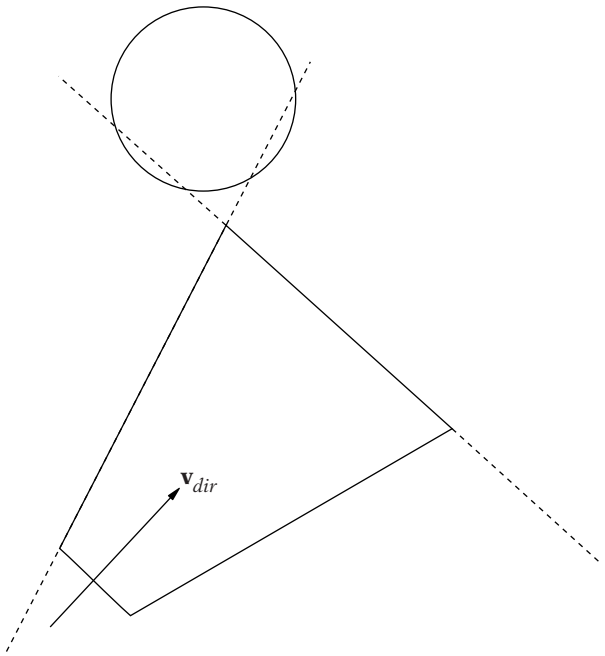


FIGURE 12.30 False positive for frustum intersection.

When handling frustum culling, the most basic approach involves testing an object against the six frustum planes. If, after this test, we determine that the object lies outside one of the planes, then we consider it outside the frustum and do not render it. As with ray casting, we can improve performance by using a bounding hierarchy at progressive levels to remove obvious cases. We can also use a spatial partition again, and consider only objects that lie in the areas of the partition within the view frustum.

However, there is one aspect of frustum culling of which we need to be careful. This also applies to any intersection test that requires determining whether we are inside a convex object. Consider the situation shown in Figure 12.30. The bounding sphere is near the corner of the view frustum and clearly intersecting two planes. By using the scheme described, this sphere would be considered as intersecting the frustum, but it is clearly not. An alternative is shown in Figure 12.31(a). Instead of using the frustum, we trace around the frustum with the bounding sphere to get a rounded, larger frustum.[1] This represents the maximum extent that a bounding sphere can have and still be inside the frustum. Instead of testing the sphere, we can test

1. This process is also known as convolution.

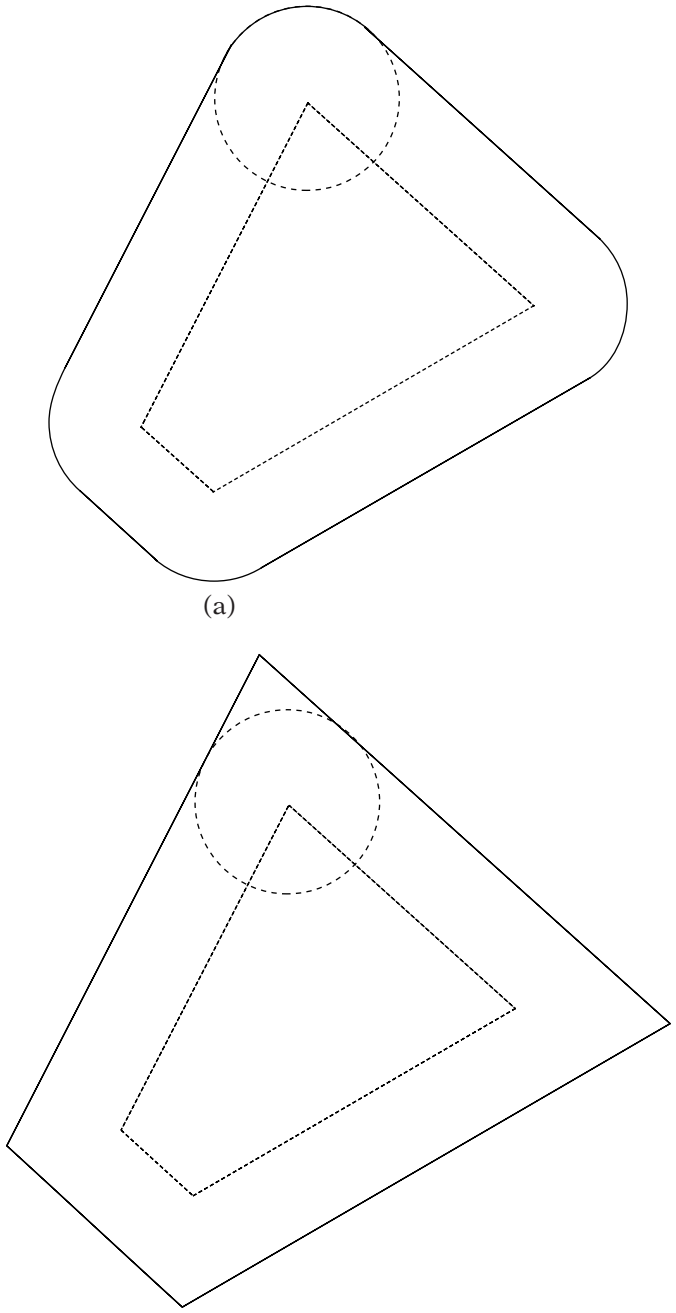


FIGURE 12.31 (a) Expanding view frustum for simpler inclusion test, and (b) expanding view frustum for simpler inclusion test.

its center against this shape. In practice, we can just push out the frustum planes by the sphere radius (Figure 12.31(b)), which is close enough. Similar techniques can be used for other bounding objects; see Akenine-Möller and Haines [82] and Watt and Policarpo [117] for more details.

12.4.6 Section Summary

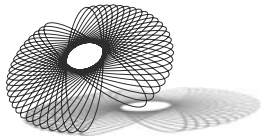
The proceeding material should give some sense of the decisions that have to be made when handling collision detection or other systems that involve object intersection: Pick base primitives, choose when you'll use them, consider whether to manage dynamic intersections, and cull unnecessary tests. However, this shouldn't be taken as the only approach. There are many other possible algorithms that handle much more complex cases than these. For example, there are systems, such as the University of North Carolina's I-COLLIDE, that track closest pairs of objects. This allows for considerable culling of intersection tests. There are also more sophisticated methods for managing spatial partitions, such as portals, octrees, BSP trees, and *kd* trees. Whether the algorithmic complexity is necessary will depend on the application.

12.5 Chapter Summary

Testing intersection between geometric primitives is a standard part of any interactive application. This chapter has presented a few examples to provide a taste of how such algorithms are created. Most derive from a careful use of the basic properties of vectors and points as presented in Chapter 2. Using our intersection methods wisely allows us to build an efficient system for detecting collision between objects, casting rays for AI visibility checks and picking, and frustum culling.

For those who are interested in reading further, a more thorough presentation of geometric distance and intersection methods can be found in Schneider and Eberly [100]. These techniques fall under a general class of algorithms known as computational geometry; good references are Preparata and Shamos [95] and O'Rourke [86]. Two different approaches to building collision-detection systems can be found in van den Bergen [112] and Ericson [32]. Finally, use of intersection techniques in rendering, plus information on more complex spatial-partitioning techniques, can be found in both Möller and Haines [82] and Watt and Policarpo [117].

This page intentionally left blank

Chapter 13

Rigid Body Dynamics

13.1 Introduction

In many games, we move our objects around using a very simple movement model. In such a game, if we hold down the up arrow key, for example, we apply a constant forward translation, once a frame, to the object until the key is released, at which point the object immediately stops moving. Similarly, we can apply a constant rotation to the object if the left arrow key is held, and again, it stops upon release. This is fine for something with fast action like a platform game or a first-person shooter, where we want quick response to our input. As soon as we hit a key, our character starts moving and stops immediately upon release. This can be thought of as an application of the theories of Aristotle, where pushing or pulling an object immediately affects its speed.

But suppose we want to do a more realistically styled game, for example, a submarine game. Submarines don't start and stop on a dime. When the propeller starts turning, it takes some time for the submarine to start forward. And they don't really have instantaneous brakes — when the engine is shut off they will drift for quite a while before stopping. Turning is much the same — they will respond slowly to application of the rudder and then straighten out over time.

Even in a fast-action game, we may want to model how objects in the world react to our main character. When we push an object, we don't expect it to stop instantly when we stop pushing, nor do we expect it to keep moving forever. If we knock a chair over, we don't expect it to fall straight back and then stick to the floor; we expect it to turn depending on where we hit it, and

then bounce and possibly roll once. We want the game world to react to our character as the real world reacts to us, in a physically correct manner.

For both of these cases, we will want a better model of movement, known as a physically based simulation. One chapter is hardly enough space to encompass this broad topic, which covers the preceding effects as well as objects deforming due to contact, fluid simulation, and soft-body simulations such as cloth and rope. Instead, we'll concentrate on a simplified problem that is useful in many circumstances: objects that don't deform (known as *rigid bodies*) and move based on Newton's laws of motion (known as *dynamics*). We'll discuss techniques for translating rigid bodies through space in a physically based manner (linear dynamics) and then how to encompass rotational effects (rotational dynamics). Finally, we'll discuss some methods for resolving contacts and dealing with simple constrained movement within our simulation, again covering linear and rotational effects in turn.

The convention in physics is to represent some vector quantities by capital letters. To maintain compatibility with physics texts we will use the same notation and assume that the reader can distinguish between such quantities and the occasional matrix by context.

13.2 Linear Dynamics

13.2.1 Moving with Constant Acceleration

Let's consider our object's movement through our game world as a function $X(t)$, which represents the position of the object for every time t. If we plot just the x values against t for the simple motion model described above, we would end up with a graph similar to that in Figure 13.1. Notice that we travel in a straight line for a while and then turn sharply in another direction, or we hold position. This is like our piecewise linear interpolation, except that in this case, the future x values are unknown; they are determined by the input of the player. For a given frame i, this can be represented by a line equation

$$X_i(h_i) = X_i + h_i \mathbf{v}_i$$

where X_i represents the position at the start of frame i, $\mathbf{v}_i$ is a vector generated from the player input that points along each line segment, and h_i is our frame time. We'll simplify things further by considering just the function on the first line segment, from time $t \geq 0$:

$$X(t) = X_0 + t\mathbf{v}_0$$

where $X_0 = X(0)$.

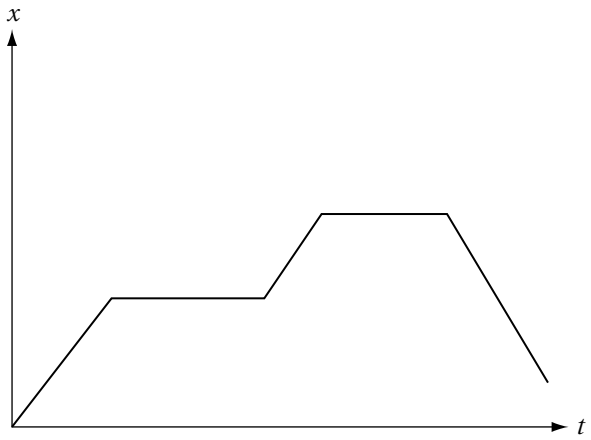


FIGURE 13.1 Graph of current motion model, showing x coordinate of particle as a function of time.

If we take the derivative of this function with respect to t, we end up with

$$\frac{d\,\mathbf{X}}{dt} = \mathbf{X}'(t) = \mathbf{v}_0 \tag{13.1}$$

This derivative of the position function is known as *velocity*, which is usually measured in meters per second, or m/s. For our simple motion model, we have a constant velocity across each segment. If we continue taking derivatives, we find that the second derivative of our position function is zero, which is what we'd expect when our velocity is constant. As mentioned, this motion model is known as *kinematics*.

Now let's assume that our second derivative, instead of being zero, is a constant nonzero function. To achieve this, we'll change our velocity function to

$$\mathbf{v}(t) = \mathbf{v}_0 + t\mathbf{a} \tag{13.2}$$

Now $\mathbf{v}(t)$ is also an affine function, this time with a constant derivative vector $\mathbf{a}$, called acceleration, or

$$\frac{d\,\mathbf{v}}{dt} = \mathbf{v}'(t) = \mathbf{a} \tag{13.3}$$

The units for acceleration are usually measured in meters per second squared, or m/s^2.

Our original function $X(t)$ used a constant $\mathbf{v}_0$, so now we'll need to rewrite it in terms of $\mathbf{v}(t)$. Since $\mathbf{v}$ is changing at a constant rate across our time

interval, we can instead use the average velocity across the interval, which is just one-half the starting velocity plus the ending velocity, or

$$\bar{\mathbf{v}} = \frac{1}{2}(\mathbf{v}_0 + \mathbf{v}(t))$$

Substituting this into our original $X(t)$ gives us

$$X(t) = X_0 + t\left[\frac{1}{2}(\mathbf{v}_0 + \mathbf{v}(t))\right]$$

Substituting in for $\mathbf{v}(t)$ gives the final result of

$$X(t) = X_0 + t\,\mathbf{v}_0 + \frac{1}{2}t^2\mathbf{a} \tag{13.4}$$

Our equation for position becomes a quadratic equation, and our velocity is represented as a linear equation:

$$P_i(t) = P_i + t\mathbf{v}_i + \frac{1}{2}t^2\mathbf{a}_i$$
$$\mathbf{v}_i(t) = \mathbf{v}_i + t\mathbf{a}_i$$

So, given a starting position and velocity and an acceleration that is constant over the entire interval $[0, t]$, we can compute any position within the interval. As an example, let's suppose we have a projectile, with an initial velocity $\mathbf{v}_0$ and initial position P_0. We represent acceleration due to gravity by the constant g, which is 9.8 m/s^2. This acceleration is applied only downward, or in the $-z$ direction, so $\mathbf{a}$ is the vector $(0, 0, -g)$. If we plot the z component as a function of t, then we get a parabolic arc, as seen in Figure 13.2. This function will work for any projectile (assuming we ignore air friction), from

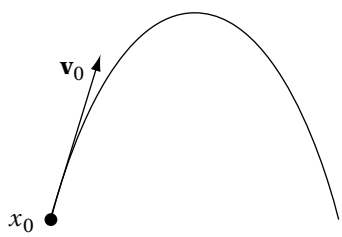


Figure 13.2 Parabolic path of object with initial velocity and affected only by gravity.

a thrown rock (low initial velocity) to a cannonball (medium initial velocity) to a bullet (high initial velocity).[1]

Within our game, we can use these equations on a frame-by-frame basis to compute the position and velocity at each frame, where the time between frames is h_i. So, for a given frame $i + 1$,

$$X_{i+1} = X_i + h_i \mathbf{v}_i + \frac{1}{2} h_i^2 \mathbf{a}_i$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + h_i \mathbf{a}_i$$

This process of motion with nonzero acceleration is known as *dynamics*.

13.2.2 Forces

One question that has been left open is how to compute our acceleration value. We do so based on a vector quantity known as a *force*. Forces cause change in an object's motion, pushing or pulling it around, either to speed it up or slow it down. So for example, to throw a ball your hand and arm exert a certain force on it, to begin its motion through the air. That force, when applied, produces an acceleration directly proportional to the object's mass, measured in kilograms. The proportional relationship is shown in Newton's second law of motion:

$$\mathbf{F} = m\,\mathbf{a}$$

The units for force end up being $kg\text{-}m/s^2$, or newtons, in homage to its creator.

In the previous section we represented gravity as an acceleration, but in truth it is a force whose value is always proportional to the mass of the object. For an object with mass m on Earth, its magnitude is mg and its direction points to the center of the Earth. In games and other small-scale simulations, we usually assume the world is locally flat and so the gravity vector points in the $-z$ direction. Other possible forces include the friction caused by air or water molecules pushing against an object to slow it down, or the thrust generated by a rocket engine or propeller, or simply the *normal force* of the ground pushing up to counteract gravity (there has to be such a force, otherwise we'd sink into the earth). In general, if something is pushing or pulling on an object, there is a force there.

Usually we have more than one force applied to an object at a time. Taking our ball example, we have the initial force when the ball is thrown, force due to gravity, and forces due to air resistance and wind. After the ball

1. In most cases, this last is approximated by a line equation for efficiency reasons.

leaves your hand, that pushing force will be removed, leaving only gravity and air effects. Forces are vectors, so in both cases we can add all forces on an object together to create a single force that encapsulates their total effect on the object. We then scale the total force by $1/m$ to get the acceleration for equation 13.4.

For simplicity's sake, we will assume for now that our forces are applied in such a way that we have no rotational effects. In Section 13.4 we'll discuss how to handle such cases.

13.2.3 Linear Momentum

As we've seen, the relationship between acceleration and velocity is

$$\mathbf{a} = \frac{d\mathbf{v}}{dt}$$

There is a corresponding related entity $\mathbf{P}$ for a force $\mathbf{F}$, which is

$$\mathbf{F} = m\mathbf{a} = m\frac{d\mathbf{v}}{dt} = \frac{d\mathbf{P}}{dt}$$

The quantity $\mathbf{P} = m\mathbf{v}$ is known as the *linear momentum* of the object, and it represents the tendency for an object to remain in its current linear motion. The heavier the object or faster it is moving, the greater the force needed to change its velocity. So, while a pebble at rest is easier to kick aside than a boulder, this is not necessarily true if the pebble is shot out of a gun.

An important property of Newtonian physics is the conservation of momentum. Suppose we take a collection of objects and treat them as a single system of objects. Now consider only the forces within the system; that is, only those forces acting between objects. Newton's third law of motion states that for every action, there is an equal and opposite reaction. So for example, if you push on the ground due to gravity, the ground pushes back just as much, and the forces cancel. Due to this, within the system, pairwise forces between objects will cancel and the total force is zero. If the external force is 0 as well, then

$$\mathbf{F} = \frac{d\mathbf{P}}{dt} = 0$$

so $\mathbf{P}$ is constant. No matter how objects may move within the system, the total momentum must be conserved. This property will be useful to us when we consider collisions.

13.2.4 Moving with Variable Acceleration

There is a problem with the approach that we've been taking so far: We are assuming that total force, and hence acceleration, is constant across the entire interval. For more complex simulations this is not the case. For example, it is common to compute a drag force proportional to but opposite in direction to velocity:

$$\mathbf{F}_{drag} = -m\rho\mathbf{v} \tag{13.5}$$

This can provide a simple approximation to air friction; the faster we go, the greater the friction force. The quantity ρ in this case controls the magnitude of drag. An alternative example is if we wish to model a spring in our system. The force applied depends on the current length of the spring, so the force is dependent on position:

$$\mathbf{F}_{spring} = -kX$$

The spring constant k fulfills a similar role to ρ: It controls the proportion of force dependent on the position. In both of these cases, since acceleration is directly dependent on the force, it will vary over the time interval as velocity or position vary. It is no longer constant. So for these cases, equations 13.2 and 13.4 are incorrect.

In order to handle this, we'll have to use an alternative approach. We begin by deriving a function for velocity in terms of any acceleration. Rewriting equation 13.3 gives us

$$d\mathbf{v} = \mathbf{a}\,dt$$

To find $\mathbf{v}$ we take the indefinite integral or antiderivative of both sides:

$$\int d\mathbf{v} = \int \mathbf{a}\,dt$$

For example, if we assume as before that $\mathbf{a}$ is constant, we can move it outside the integral sign:

$$\int d\mathbf{v} = \mathbf{a}\int dt$$

And integrating gives us

$$\mathbf{v} = t\mathbf{a} + \mathbf{c}$$

We can solve for $\mathbf{c}$ by using our velocity $\mathbf{v}_0$ at time $t = 0$:

$$\begin{aligned}\mathbf{c} &= \mathbf{v}_0 - 0 \cdot \mathbf{a} \\ &= \mathbf{v}_0\end{aligned}$$

So, our final equation is as before:

$$\mathbf{v}(t) = \mathbf{v}_0 + t\,\mathbf{a}$$

We can perform a similar integration for position. Rewriting equation 13.1 gives

$$dX = \mathbf{v}(t)dt$$

We can substitute equation 13.2 into this to get

$$dX = \mathbf{v}_0 + t\,\mathbf{a}\,dt$$

Integrating this, as we did with velocity, produces equation 13.4 again.

For general equations we perform the same process, reintegrating $d\mathbf{v}$ to solve for $\mathbf{v}(t)$ in terms of $\mathbf{a}(t)$. So, using our drag example, we can divide equation 13.5 by the mass m to give acceleration:

$$\mathbf{a} = \frac{d\,\mathbf{v}}{dt} = -\rho\,\mathbf{v}(t)$$

Rearranging this and integrating gives

$$\int d\,\mathbf{v} = \int -\rho\,\mathbf{v}(t)dt$$

We can consult a standard table of integrals to find that the answer in this case is

$$\mathbf{v}(t) = \mathbf{v}_0 e^{-\rho t}$$

where, as before, $\mathbf{v}_0 = \mathbf{v}(0)$.

While this particular equation was relatively straightforward, in general calculating an exact solution is not as simple as the case of constant acceleration. First of all, differential equations in which the quantity we're solving for is part of the equation are not always easily — if at all — solvable by analytic means. In many cases, we will not necessarily be able to find an exact equation for $\mathbf{v}(t)$, and thus not for $X(t)$. And even if we can find a solution, every time we change our simulation equations, we'll have to integrate them

again, and modify our simulation code accordingly. Since we'll most likely have many different possible situations with many different applications of force, this could grow to be quite a nuisance. Because of both these reasons, we'll have to use a numerical method that can approximate the result of the integration.

13.3 Numerical Integration

13.3.1 Definition

The solutions for $\mathbf{v}$ and X that we're trying to integrate fall under a class of differential equation problems called *initial value problems*. In an initial value problem, we know the following about a function $\mathbf{y}(t)$:

1. An initial value of the function $\mathbf{y}_0 = \mathbf{y}(t_0)$.
2. A derivative function $\mathbf{f}(t, \mathbf{y}) = \mathbf{y}'(t)$.
3. A time interval h.

The problem we're trying to solve is, given these parameters, what is the value at $\mathbf{y}(t_0 + h)$? For our purposes, this actually becomes a series of initial value problems: At each frame our previous solution becomes our new initial value $\mathbf{y}_i$, and our interval h_i will be based on the current frame time. Once computed, our new solution will become the next initial value $\mathbf{y}_{i+1}$. More specifically, the initial value $\mathbf{y}_i$ is our current position X_i and current velocity $\mathbf{v}_i$, stored in a single 6-vector as

$$\mathbf{y}_i = \begin{bmatrix} X_i \\ \mathbf{v}_i \end{bmatrix}$$

So, how do we evaluate the derivative function $\mathbf{f}(t, \mathbf{y})$? This will be another vector quantity:

$$\mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \mathbf{X}_i' \\ \mathbf{v}_i' \end{bmatrix}$$

The value of our derivative for X_i is our current velocity $\mathbf{v}_i$. Our derivative for $\mathbf{v}_i$ is the acceleration, which is based on the current total force. To compute this total force, it is convenient to create a function called `CurrentForce()`, which takes X and $\mathbf{v}$ as arguments and combines any forces derived from

position and velocity with any constant forces, such as those created from player input. We'll represent this as $\mathbf{F}_{tot}(t, X, \mathbf{v})$ in our equations. So, given our current state, the result of our function $\mathbf{f}(t, \mathbf{y})$ will be

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}) = \begin{bmatrix} \mathbf{v}_i \\ \mathbf{F}_{tot}(t_i, X_i, \mathbf{v}_i)/m \end{bmatrix}$$

The function $\mathbf{f}(t, \mathbf{y})$ is important in understanding how we can solve this problem. For every point $\mathbf{y}$ it returns a derivative $\mathbf{y}'$. This represents a vector field, where every point has a corresponding associated vector. To get a sense of what this looks like, let's take as an example a planet rotating in a perfectly circular orbit. Figure 13.3 shows a two-dimensional (2D) plot of the vector field of position and velocity, accentuating certain lines of flow. If we start at a particular point and follow the vector flow, this will trace out one possible solution to the differential equation, starting at that initial value.

This gives us a sense of what our general approach will be. We'll start at $\mathbf{y}_i$ and then, using our derivative function, take steps in time to generate new samples that approximate the function, until we generate an approximation for $\mathbf{y}_{i+1}$. In a way, we are doing the opposite of what we were doing when we were interpolating. Instead of generating an approximation to an unknown function based on known sample points, we're generating approximate sample points based on the derivative of an unknown function. Different integration techniques are different forms of this approach, some more accurate than others.

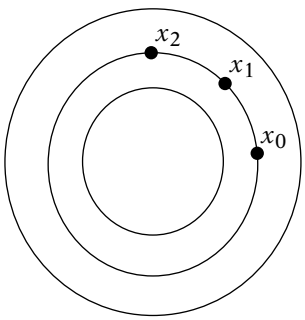


Figure 13.3 Orbit example, showing some level curves and idealized integration path.

13.3.2 Euler's Method

Source Code
Demo
Force

Assuming our current time is t and we want to move ahead h in time, we could use Taylor's series to compute $\mathbf{y}(t+h)$:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h\mathbf{y}'(t) + \frac{h^2}{2}\mathbf{y}''(t) + \cdots + \frac{h^n}{n!}\mathbf{y}^{(n)}(t) + \cdots$$

We can rewrite this to compute the value for time step $i+1$, where the time from t_i to t_{i+1} is h_i:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i\mathbf{y}_i' + \frac{h_i^2}{2}\mathbf{y}_i'' + \cdots + \frac{h_i^n}{n!}\mathbf{y}_i^{(n)} + \cdots$$

This assumes, of course, that we know all the values for the entire infinite series at time step i, which we don't — we have only $\mathbf{y}_i$ and $\mathbf{y}_i'$. However, if h_i is small enough and all values of $\mathbf{y}_i''$ are bounded, we can use an approximation instead:

$$\begin{aligned}\mathbf{y}_{i+1} &\approx \mathbf{y}_i + h_i\mathbf{y}_i' \\ &\approx \mathbf{y}_i + h_i\mathbf{f}(t_i, \mathbf{y}_i)\end{aligned}$$

Another way to think of this is that we have a function $\mathbf{f}(t_i, \mathbf{y}_i)$ that, given a time t_i and initial value $\mathbf{y}_i$, can compute tangents to the unknown function's curve. We can start at our known initial value, and step h_i distance along the tangent vector to get to the next approximation point in the vector field (Figure 13.4).

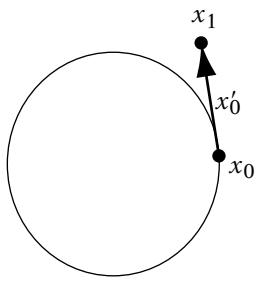


Figure 13.4 Orbit example, showing Euler step.

Separating out position and velocity gives us

$$\begin{aligned} X_{i+1} &\approx X_i + h_i X_i' \\ &\approx X_i + h_i \mathbf{v}_i \\ \mathbf{v}_{i+1} &\approx \mathbf{v}_i + h_i \mathbf{v}_i' \\ &\approx \mathbf{v}_i + h_i \mathbf{F}_{tot}(t_i, X_i, \mathbf{v}_i)/m \end{aligned}$$

This is known as *Euler's method*.

To use this in our game, we start with our initial position and velocity. At each new frame, we grab the difference in time between the previous frame and current frame and use that as h_i. To compute $\mathbf{f}(t_i, \mathbf{y}_i)$ for the velocity, we use our `CurrentForce()` method to add up all of the forces on our object and divide the result by the mass to get our acceleration. Plugging in our current values, we use the preceding formulas to generate our new position and velocity. In code, this looks like the following.

```
void
SimObject::Integrate( float h )
{
    IvVector3 accel;

    // compute acceleration
    accel = CurrentForce( mTime, mPosition, mVelocity ) / mMass;
    // clear small values
    accel.Clean();

    // compute new position, velocity
    mPosition += h*mVelocity;
    mVelocity += h*accel;
    // clear small values
    mVelocity.Clean();
}
```

It's important to compute the new velocity after the new position in this case, so that we don't overwrite the velocity prematurely.

Note that we clear near-zero values in the new velocity. This prevents little shifts in position due to tiny changes in velocity, such as those generated after an object has slowed down due to drag. While technically accurate, they can be visually distracting, so after a certain point we clamp our velocity to zero. The same is done with acceleration.

For many cases, this works quite well. If our time steps are small enough, then the resulting approximation points will lie close to the actual function and we will get good results. However, the ultimate success of this method is based on the assumption that the slope at the current point is a good estimate of the slope over the entire time interval h. If not, then the approximation can drift off the function, and the farther it drifts, the worse the tangent approximation can get. We can see this with our orbit example in Figure 13.5. The first step in our approximation takes us to an orbit with a larger radius, and the next step to a larger radius still. Once the error grows, in many cases further steps don't get us back, and we continue to drift off of the actual solution.

For Euler's method, we say that the error is directly dependent on the time step, or $O(h)$. So, one potential solution to this problem is to decrease the time step, for example, take a step of $h/2$, followed by another step of $h/2$. While this may solve some cases, we may need to take a smaller time step, say $h/4$. And this may still lead to significant error. In the meantime, we are grinding our simulation to a halt while we recalculate quantities 4 or 8, or however many times for a single frame.

So, what's happening here? First, some situations that can lead to problems with Euler's method are characterized by large forces. If we examine the remaining terms of the Taylor expansion,

$$\frac{h_i^2}{2}\mathbf{y}_i'' + \cdots + \frac{h_i^n}{n!}\mathbf{y}_i^{(n)} + \cdots$$

we can see why this could cause a problem. When we set up our approximation, we assumed that h_i was small and $\mathbf{y}_i''$ bounded. If we're considering

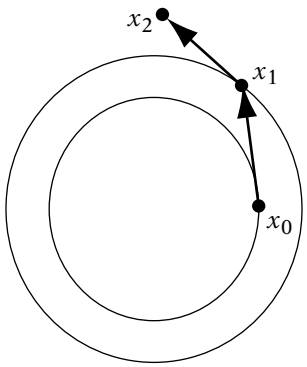


Figure 13.5 Orbit example, showing continuation of Euler's method.

position, a large force leads to a large acceleration, which leads to a larger difference between our approximation and the actual value. Larger values of h_i will magnify this error. Also, if the force changes quickly, this means that the magnitude of the velocity's second derivative is high, and so we can run into similar problems with velocity. This is known as truncation error, and as we can see, with Euler's method the truncation error is $O(h^2)$.

However, our particular example falls into a class of differential equations known as *stiff* systems. Situations that can lead to stiffness problems are often characterized by large spring and damping forces, such as in a stiff spring (hence the name). Examples of such systems have terms with rapidly decaying values, such as $e^{-\rho t}$ — exactly the case when we are trying to maintain a fixed distance from a point. These terms tend to zero as t approaches infinity but, as we've seen, won't always converge with a numerical method. The larger ρ is, the smaller h must be. This can also affect systems where we wouldn't expect the term to contribute that much. For example, suppose the solution to our system is $y(t) = 1 + e^{-200t}$. As t increases from zero, $y(t)$ quickly approaches 1. However, approximating this with a numerical method without taking care to control the error can lead the e^{-200t} term to dominate the calculations, which leads to invalid results.

Due to these issues, Euler's method is not a very robust integrator. It is, however, quite cheap and easy to implement, which is why a lot of simple physics engines use it. Fortunately, there are other methods that we can try.

13.3.3 Runge-Kutta Methods

Source Code DEMO Force

So far we've been using the derivative at the beginning of the interval as our estimate of the average tangent. A better possibility may be to take the derivative in the middle of the interval. To do this, we first use Euler's method to take a step halfway into the interval; that is, we integrate using a step size of $h/2$. Given our estimated position and velocity at the halfway point, we calculate $\mathbf{f}(t, \mathbf{y})$ at this location. We then go back to our original starting location, and use the derivatives we calculated at the midpoint to move across the entire interval. This method is known as the *midpoint method*.

Figure 13.6 show how this works with our original function. In Figure 13.6(a), the arrow shows our initial half-step, and the line our estimated tangent. Figure 13.6(b) uses the tangent we've calculated with our full time step, and our final location. As we can see, with this method we are following much closer to the actual solution and so our error is much less than before. The order of the error for the midpoint method is dependent on the square of the time step, or $O(h^2)$, which for values of h less than 1 is better

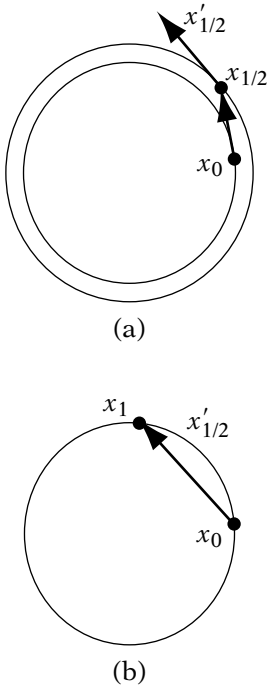


FIGURE 13.6 (a) Orbit example, showing first step of midpoint method: getting the midpoint derivative. (b) Orbit example, stepping with midpoint derivative to next estimate.

than Euler's method. Instead of approximating the function with a line, we are approximating it with a quadratic.

While the midpoint method does have better error tolerance than Euler's method, as we can see from our example, it still drifts off of the desired solution. To handle this, we'll have to consider some methods with better error tolerances still.

Both the midpoint method and Euler's method fall under a larger class of algorithms known as *Runge-Kutta methods*. Whereas both of our previous techniques used a single estimate to compute a tangent for the entire interval, others within the Runge-Kutta family compute multiple tangents at fixed time steps across the interval and take their weighted average.

One possibility is to take the derivative at the end of the interval, and average with the derivative at the beginning. Like the midpoint method, we can't actually compute the derivative at the end of the interval, so we'll approximate it by performing normal Euler integration and computing the derivative at that point. This is known as the *modified Euler's method*. Interestingly, the

error for this approach is still $O(h^2)$, due to the fact that we're taking an inaccurate measure of the final derivative. Another approach is *Heun's method*, which takes 1/4 of the starting derivative, and 3/4 of an approximated derivative 2/3 along the step size. Again, its error is $O(h^2)$, or no better than the midpoint method.

The standard $O(h^4)$ method is known as *Runge-Kutta order four*, or simply RK4. RK4 can be thought of as a combination of the midpoint method and modified Euler, where we weight the midpoint tangent estimates higher than the endpoint estimates. Representing this with our function notation, we get

$$\begin{aligned}
\mathbf{u}_1 &= h_i \mathbf{f}(t_i, \mathbf{y}_i) \\
\mathbf{u}_2 &= h_i \mathbf{f}(t_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{1}{2}\mathbf{u}_1) \\
\mathbf{u}_3 &= h_i \mathbf{f}(t_i + \frac{h_i}{2}, \mathbf{y}_i + \frac{1}{2}\mathbf{u}_2) \\
\mathbf{u}_4 &= h_i \mathbf{f}(t_i + h_i, \mathbf{y}_i + \mathbf{u}_3) \\
\mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{6}[\mathbf{u}_1 + 2\mathbf{u}_2 + 2\mathbf{u}_3 + \mathbf{u}_4]
\end{aligned}$$

Clearly, improved accuracy doesn't come without cost. To perform standard Euler requires calculating a result for $f(t, \mathbf{y})$ only once. Midpoint, modified Euler, and Heun's need two calculations, and RK4 takes four. While achieving the level of error tolerance of RK4 would require many more evaluations of Euler's method, using RK4 still adds both complexity and increased simulation time that may not be necessary. It does depend on your application, but for simple rigid-body simulations with fast frame rates and low accelerations, Euler's method or one of the other two Runge-Kutta methods will probably be suitable.

13.3.4 Verlet Integration

There is another class of integration methods, known as *Verlet methods*, that is commonly used in molecular dynamics. Verlet methods have come to the attention of the games community because they can be useful in simulating collections of small, unoriented masses known as particles—in particular, when constrained distances between particles are required [61]. Such systems of constrained particles can simulate soft objects such as cloth, rope, and dead bodies (this last one is also known as rag doll physics).

The most basic Verlet method can be derived by adding the Taylor expansion for the current time step with the expansion for the previous time step:

$$\mathbf{y}(t+h) + \mathbf{y}(t-h) = \mathbf{y}(t) + h\mathbf{y}'(t) + \frac{h^2}{2}\mathbf{y}''(t) + \cdots$$
$$+ \mathbf{y}(t) - h\mathbf{y}'(t) + \frac{h^2}{2}\mathbf{y}''(t) - \cdots$$

Solving for $\mathbf{y}(t+h)$ gives us

$$\mathbf{y}(t+h) = 2\mathbf{y}(t) - \mathbf{y}(t-h) + h^2\mathbf{y}''(t) + O(h^4)$$

Rewriting in our stepwise format, we get

$$\mathbf{y}_{i+1} = 2\mathbf{y}_i - \mathbf{y}_{i-1} + h_i^2\mathbf{y}_i''$$

This gives us an $O(h^2)$ solution for integrating position from acceleration, without involving velocity at all. This can be a problem if we want to use velocity in our calculations, but we can estimate it as

$$\mathbf{v}_i = \frac{(X_{i+1} - X_i)}{2h_i}$$

One question may be, how do we find the first $\mathbf{y}_{i-1}$? The standard method is to start the process off with one pass of standard Euler or other Runge-Kutta method and store the initial position and integrated position. From there we'll have two positions to apply to our Verlet integration.

Standard Verlet has a few advantages: It is time invariant, which means that we can run it forwards and then backwards and end up in the same place. Also, the lack of velocity means that we have one less quantity to calculate. Because of this, it is often used for particle systems, which generally are not dependent on velocity. However, if we want to apply friction based on velocity or when we want to handle spinning rigid objects, the lack of velocity and angular velocity makes it more difficult. There are ways around this, as described in Jakobson [61], but in most cases it will be easier to use a method that allows us to track both velocity terms. One other disadvantage is that our velocity estimation is (a) not very accurate and (b) one time step behind our position.

If you wish to use Verlet methods and require velocity, you have two choices. *Leapfrog Verlet* tracks velocity, but at half a time step off from the

position calculation:

$$\mathbf{v}(t + h/2) = \mathbf{v}(t - h/2) + h\mathbf{a}(t)$$
$$X(t + h) = X(t) + h\,\mathbf{v}(t + h/2)$$

Like with standard Verlet, we can start this off with a Runge-Kutta method by computing velocity at a half-step and proceed from there. If velocity on a whole step is required, it can be computed from the velocities, but as with standard Verlet, one time step behind position:

$$\mathbf{v}_i = \frac{(\mathbf{v}_{i+1/2} - \mathbf{v}_{i-1/2})}{2}$$

As with standard Verlet, leapfrog Verlet is an $O(h^2)$ method.

The third, and most accurate, Verlet method is *velocity Verlet*:

$$X(t + h) = X(t) + h\,\mathbf{v}(t) + \frac{h^2}{2}\mathbf{a}(t)$$
$$\mathbf{v}(t + h) = \mathbf{v}(t) + h/2[\mathbf{a}(t) + \mathbf{a}(t + h)]$$

Unlike with the previous Verlet methods, we now have to compute the acceleration twice: once at the start of the interval and once at the end. This can be done in a stepwise manner by

$$\mathbf{v}_{i+1/2} = \mathbf{v}_i + h_i/2\,\mathbf{a}_i$$
$$X_{i+1} = X_i + h_i\,\mathbf{v}_{i+1/2}$$
$$\mathbf{v}_{i+1} = \mathbf{v}_{i+1/2} + h_i/2\,\mathbf{a}_{i+1}$$

In between the position calculation and the velocity calculation, we recompute our forces and then the acceleration $\mathbf{a}_{i+1}$. Note that in this case the forces can be dependent only on position, since we have added only half of the acceleration contribution to velocity. In the case of molecular dynamics or particles, this isn't a problem since most of the forces between them will be positional, but again, for rigid-body problems this is not the case.

While Verlet integration has good stability characteristics, its main problem for our purposes is the estimated velocity, as mentioned above. While it works well for particle systems, it isn't as good for rigid bodies. As such, we'll look elsewhere for our solution.

13.3.5 Implicit Methods

All the methods we've described so far integrate based on the current position and velocity. They are called *explicit methods* and make use of known quantities at each time step, for example Euler's method:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\,\mathbf{y}'_i$$

But as we've seen, even higher-order explicit methods don't handle extreme cases of stiff equations very well.

Implicit methods make use of quantities from the next time step:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i\,\mathbf{y}'_{i+1}$$

This particular implicit method is known as *backward Euler*. The idea is that we are going to grab the derivative at our destination rather than at our current position. That is, we are going to find a $\mathbf{y}_{i+1}$ with the derivative that, if we were to run the simulation backwards, would end up at $\mathbf{y}_i$.

Implicit methods don't add energy to the system, but instead lose it. This doesn't guarantee us more accuracy, but it does avoid simulations that spin out of control — instead, they'll dampen down to an equilibrium state. Since, in most cases, we're going to add a damping factor anyway, this is a small price to pay for a more stable simulation. An example of using this is our old orbit example (Figure 13.7). Here we see the effect of losing energy — instead of spiraling outward, we spiral inward toward the center of the orbit. Better than Euler's method, but still not ideal.

This sounds good in theory, but in practice, how do we calculate $\mathbf{y}'_{i+1}$? One way is to solve for it directly. For example, let's consider air friction. In

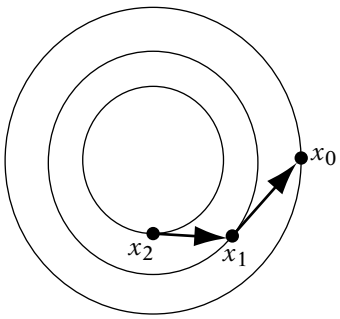


Figure 13.7 Implicit Euler. The arrows point backwards to indicate that we are getting the derivative from the next time step.

this example, our force is directly dependent on velocity, but in the opposing direction. Considering only velocity:

$$\mathbf{v}_{i+1} = \mathbf{v}_i - h\rho\mathbf{v}_{i+1}$$

Solving for $\mathbf{v}_{i+1}$ gives us

$$\mathbf{v}_{i+1} = \frac{\mathbf{v}_i}{1 + h\rho}$$

We can't always use this approach. Either we will have a function too complex to solve in this manner, or we'll be experimenting with a number of functions and won't want to take the time to solve each one individually. Another way is to use a *predictor–corrector* method. We move ahead one step using an explicit method to get an approximation. Then we use that approximation to calculate our $\mathbf{y}'_{i+1}$. This will be more accurate than the explicit method alone, but it does involve twice the number of calculations, and we're depending on the accuracy of the first approximation to make our final calculation.

Another more accurate approach is to rewrite the equation so that it can be solved as a linear system. If we represent $\mathbf{y}_{i+1}$ as $\mathbf{y}_i + \Delta\mathbf{y}_i$, and ignore the factor t, we can rewrite backwards Euler as

$$\mathbf{y}_i + \Delta\mathbf{y}_i = \mathbf{y}_i + h_i\mathbf{f}(\mathbf{y}_i + \Delta\mathbf{y}_i)$$

or

$$\Delta\mathbf{y}_i = h_i\mathbf{f}(\mathbf{y}_i + \Delta\mathbf{y}_i)$$

We can approximate $\mathbf{f}(\mathbf{y}_i + \Delta\mathbf{y}_i)$ as $\mathbf{f}(\mathbf{y}_i) + \mathbf{f}'(\mathbf{y}_i)\Delta\mathbf{y}_i$. Note that $\mathbf{f}'(\mathbf{y}_i)$ is a matrix since $\mathbf{f}(\mathbf{y}_i)$ is a vector. Substituting this approximation, we get

$$\Delta\mathbf{y}_i \approx h_i(\mathbf{f}(\mathbf{y}_i) + \mathbf{f}'(\mathbf{y}_i)\Delta\mathbf{y}_i)$$

Solving for $\Delta\mathbf{y}_i$ gives

$$\Delta\mathbf{y}_i \approx \left(\frac{1}{h_i}\mathbf{I} - \mathbf{f}'(\mathbf{y}_i)\right)^{-1}\mathbf{f}(\mathbf{y}_i)$$

In most cases, this linear system will be sparse, so it can be solved in near-linear time. More information can be found in Witkin and Baraff [121].

While implicit methods do have some characteristics that we like—they're good for forces that depend on stiff equations—they do tend to lose energy and may dampen more than we might want. Again, this is better than explicit Euler, but it's not ideal. They're also more complex and more expensive than explicit Euler. Fortunately there is a solution that provides the simplicity of explicit Euler with the stability of implicit Euler.

13.3.6 Semi-Implicit Methods

Up to this point, we have been treating position and velocity as independent variables while integrating; that is, we act as if they are one six-element vector that gets integrated at once. However, the fact is that position is dependent on how velocity changes. We can make use of this relationship and create a very stable integrator for dynamics. The trick is to run an explicit Euler step for velocity, and then an implicit Euler step for position:

$$\begin{aligned}
\mathbf{v}_{i+1} &\approx \mathbf{v}_i + h_i \mathbf{v}_i' \\
&\approx \mathbf{v}_i + h_i \mathbf{F}_{tot}(t_i, X_i, \mathbf{v}_i)/m \\
X_{i+1} &\approx X_i + h_i X_i' \\
&\approx X_i + h_i \mathbf{v}_{i+1}
\end{aligned}$$

Note that the position update is using the *new* velocity, not the old one. This is called *semi-implicit* or *symplectic* Euler. Note that position is integrated using implicit Euler, which makes this particularly good for position-dependent forces. Thus, this method gives us the advantages of both explicit and implicit methods, plus it also has an additional advantage: It conserves energy over time, which keeps things very stable.

Let's look at our orbit example again, this time using semi-implicit Euler (Figure 13.8). We note that it follows the path exactly, rather than converging or diverging. Admittedly, this example is a bit contrived, but it shows the power of using a semi-implicit method.

Because it is a first-order Euler method it's still not as accurate in some cases as RK4, but it is cheap and stable. And in games, it's far more important to have a stable solution than a 100 percent correct one. This integration technique is also very easy to adapt to rotational dynamics. This makes it suitable for most of our needs beyond the most egregious cases, and thus will be the method we use for our examples.

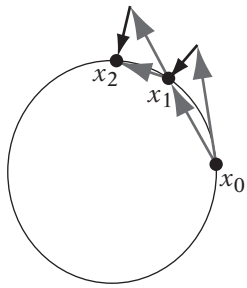


Figure 13.8 Semi-implicit Euler. The gray arrows indicate the original velocity and its modification by acceleration.

13.4 Rotational Dynamics

13.4.1 Definition

The equations and methods that we've discussed so far allow us to create physical simulations that modify an object's position. However, one aspect of dynamics we've passed over is simulating changes in an object's orientation due to the application of forces, or *rotational dynamics*. When discussing rotational dynamics, we use quantities that are very similar to those used in linear dynamics. Comparing the two:

Linear	Rotational
position X	orientation $\mathbf{\Omega}$ or $\mathbf{q}$
velocity $\mathbf{v}$	angular velocity ω
force $\mathbf{F}$	torque τ
linear momentum $\mathbf{P}$	angular momentum $\mathbf{L}$
mass m	inertia tensor $\mathbf{J}$

We'll discuss each of these quantities in turn.

13.4.2 Orientation and Angular Velocity

Orientation we have seen before; we'll represent it by a matrix $\mathbf{\Omega}$ or a quaternion $\mathbf{q}$. The angular velocity ω represents the change in orientation. It is a vector quantity, where the vector direction is the axis we rotate around to effect the change in orientation, and the length of the vector represents the rate of rotation around that axis, in radians per second.

The orientation and angular velocity are applied to an object around a point known as the *center of mass*. The center of mass can be defined as the

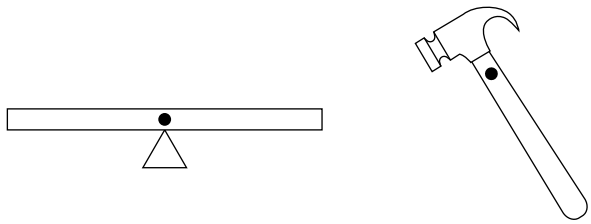


FIGURE 13.9 Comparing centers of mass. The seesaw balances close to the center, while the hammer has center of mass closer to the end.

point associated with an object where, if you apply a force at that point, it will move without rotating. One can think of it as the point where the object would perfectly balance. Figure 13.9 shows the center of mass for some common objects. The center of mass for a seesaw is directly in the center, as we'd expect. The center of mass for a hammer, however, is closer to one end than the other, since the head of the hammer is more massive than the handle.

For our objects, we'll assume that we have some sense of where the center of mass is — either it's set by the artist or by some other means. One possibility discussed shortly is to compute the center of mass directly from our model data. Other choices are to use the local model origin or the bounding box center (or centroid) as an approximation. Once the center of mass is determined, it is usually convenient to translate our object so that we can treat the local model origin as the center of mass, and therefore use the same orientation and position representation for both simulation and rendering.

It is possible to convert from angular velocity to linear velocity. Given an angular velocity ω, and a point at displacement $\mathbf{r}$ from the center of mass, we can compute the linear velocity at the point by using the equation

$$\mathbf{v} = \omega \times \mathbf{r} \tag{13.6}$$

This makes sense if we look at a rotating sphere. If we look at various points on the sphere (Figure 13.10(a)), their linear velocity is orthogonal to both the axis of rotation and their displacement vector, and this corresponds to the direction of the cross product. The length of $\mathbf{v}$ will be

$$\|\mathbf{v}\| = \|\omega\| \|\mathbf{r}\| \sin\theta$$

where θ is the angle between ω and $\mathbf{r}$. This also makes sense. As the rate of rotation $\|\omega\|$ increases, we'd expect the linear velocity of each point on the object to increase. As we move out from the equator, a rotating point has to move a longer linear distance in order to maintain the same angular velocity relative to the center (Figure 13.10(b)), so as $\|\mathbf{r}\|$ increases, $\|\mathbf{v}\|$ will increase.

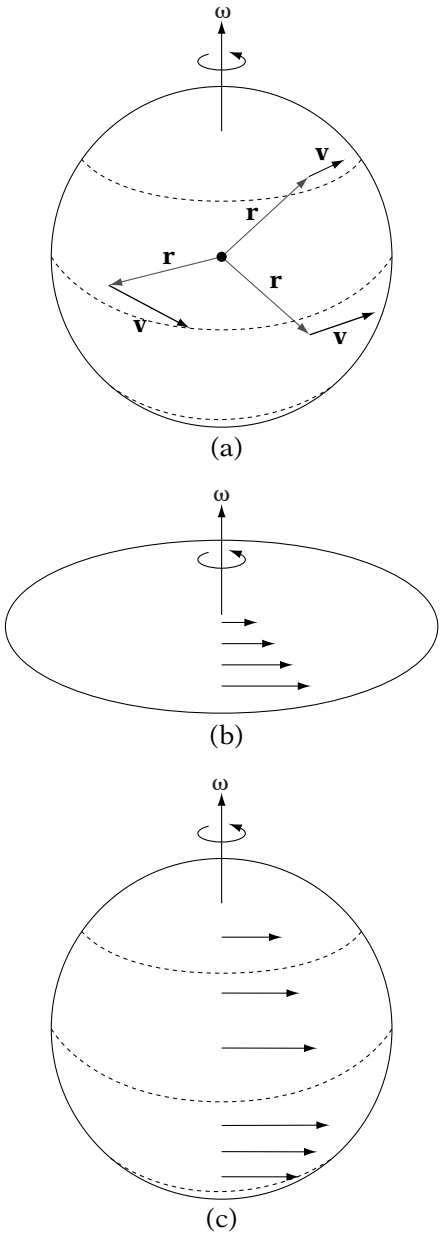


FIGURE 13.10 (a) Linear velocity of points on the surface of a rotating sphere. Velocity is orthogonal to both angular velocity vector and displacement vector from the center of rotation. (b) Comparison of speed of points on surface of rotating disk. Points further from the center of rotation have larger linear velocity. (c) Comparison of speed of points on surface of rotating sphere. Points closer to the equator of the sphere have larger linear velocity.

Finally, the linear velocity of a point as we move from the equator to the poles will decrease to zero (Figure 13.10(c)) and the quantity $\sin\theta$ provides this.

13.4.3 Torque

Up until now we've been simplifying our equations by applying forces only at the center of mass, and therefore generating only linear motion. On the other hand, if we apply an off-center force to an object, we expect it to spin. The rotational force created, known as *torque*, is directly dependent on the location where the force is applied. The farther away from the center of mass we apply a given force, the larger the torque. To compute torque, we take the cross product of the vector from the center of mass to the force application point, with the corresponding force (Figure 13.11) or

$$\tau = \mathbf{r} \times \mathbf{F} \tag{13.7}$$

The direction of τ combined with the right-hand rule tells us the direction of rotation the torque will attempt to induce. If you align your right thumb along the direction of torque, your curled fingers will indicate the direction of rotation — if the vector is pointing toward you, this is counterclockwise around the axis of torque. The magnitude of τ provides the magnitude of the corresponding torque.

To compute the total torque, we need to compute the corresponding torque for each application of force, and then add them up. Adding the offsets and taking the cross product of the resulting vector with the total force will not compute the correct result, as shown by Figure 13.12. The sum of the offsets is **0**, producing a torque of **0**, which is clearly not the case — the true total torque as shown will start the circle rotating counterclockwise.

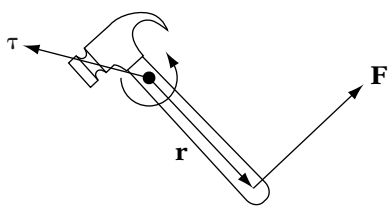


Figure 13.11 Computing torque. Torque is the cross product of displacement vector and force vector.

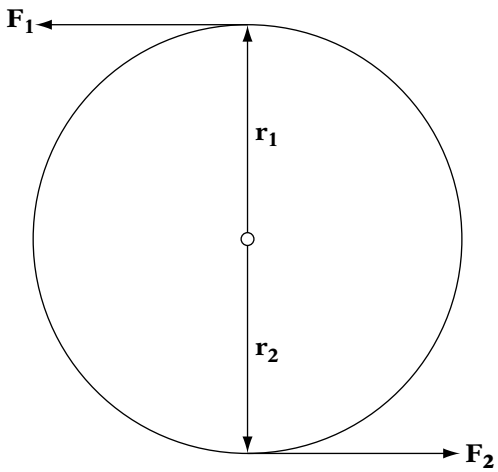


FIGURE 13.12 Adding two torques. If forces and displacements are added separately and then the cross product is taken, total torque will be 0. Each torque must be computed and then added together.

13.4.4 ANGULAR MOMENTUM AND INERTIA TENSOR

Recall that a force **F** is the derivative of the linear momentum **P**. There is a related quantity **L** for torque, such that

$$\tau = \frac{d\boldsymbol{L}}{dt}$$

Like linear momentum, the angular momentum **L** describes how much an object tends to stay in motion, but in rotational motion rather than linear motion. The higher the angular momentum, the larger the torque needed to change the object's angular velocity. Recall that linear momentum is equal to the mass of the object times its velocity. Angular momentum is similar, except that we use angular velocity, and the rotational equivalent of mass, the inertia tensor matrix:

$$\mathbf{L} = \mathbf{J}\,\omega \tag{13.8}$$

Why use a matrix **J** instead of a scalar, as we did with mass? The problem is that while shape has no effect (other than, say, for friction) on the general equations for linear dynamics, it does have an effect on how objects rotate. Take the classic example of a figure skater in a spin. As she starts the spin, her arms are out from her sides, and she has a low angular velocity. As she brings her arms in, her angular velocity increases until she opens her arms again to

gracefully pull out of the spin. Torque is near zero in this case (ignoring some minimal friction from the ice and air), so we can consider angular momentum to be constant. Since angular velocity is clearly changing and mass is constant, the shape of the skater is the only factor that has a direct effect to cause this change.

So, to represent this effect of shape on rotation, we use a 3×3 symmetric matrix, where

$$\mathbf{J} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}$$

We need these many factors because, as we've said, rotation depends heavily on shape and each factor describes how the rotation changes around a particular axis. The diagonal elements are called the *moments of inertia*. If we're in the correct coordinate frame, then the nondiagonal elements, or *products of inertia*, are zero. For such a frame, the axes are called the *principle axes*. For example, if the object is symmetric, the principle axes lie along the axes of symmetry and through the center of mass. We'll see next how to handle the case if our object is *not* in the principle axes frame.

The following are some examples of simple inertia tensors for objects with constant density and mass m:

- Sphere (radius of r):

$$\begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix}$$

- Solid cylinder (main axis aligned along x, radius r, length d):

$$\begin{bmatrix} \frac{1}{2}mr^2 & 0 & 0 \\ 0 & \frac{1}{4}mr^2 + \frac{1}{12}md^2 & 0 \\ 0 & 0 & \frac{1}{4}mr^2 + \frac{1}{12}md^2 \end{bmatrix}$$

- Box ($x_{dim} \times y_{dim} \times z_{dim}$):

$$\begin{bmatrix} \frac{1}{12}m(y_{dim}^2 + z_{dim}^2) & 0 & 0 \\ 0 & \frac{1}{12}m(x_{dim}^2 + z_{dim}^2) & 0 \\ 0 & 0 & \frac{1}{12}m(x_{dim}^2 + y_{dim}^2) \end{bmatrix}$$

For many purposes, these can be reasonable approximations. If necessary, it is possible to compute an inertia tensor and center of mass for a generalized model, assuming a constant density. A number of methods have been presented to do this, in increasing refinement [11, 27, 63, 79]. The general concept is that in order to compute these quantities we need to do a solid integral across our shape, which is a triple integral across three dimensions. If we assume constant density, then for a polytope this is equivalent to adding up tetrahedra, where each tetrahedra consists of one of the polygonal faces and a shared central point. Code to perform this operation is available at www.geometrictools.com, for those who desire it.

13.4.5 Integrating Rotational Quantities

As with linear dynamics, we use our angular velocity to update to our new orientation. Ideally, we could use Euler's method directly and compute our new orientation as

$$\mathbf{\Omega}_{i+1} = \mathbf{\Omega}_i + h\omega_i$$

However, this won't work, mainly because we are trying to combine vector and matrix quantities. What we need to do is compute a matrix that represents the derivative and use that with Euler's method.

Recall that the column vectors of a rotation matrix are three orthonormal vectors. We need to know how each vector will change with time; that is, we need the linear velocity at each vector tip. What we want to do is convert the angular velocity into a linear velocity for each of our basis vectors. We can apply equation 13.6 to each of our basis vectors to compute this, and then use the matrix generated to integrate orientation. One way would be to take the cross product of ω with each column vector, but instead we can take our three angular velocity values, and create a skew symmetric matrix $\tilde{\omega}$, where

$$\tilde{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{13.9}$$

If we multiply this by our current orientation matrix, this will take the cross product of ω with each column vector, and we end up with the derivative of orientation in matrix form. Using this with Euler's method, we end up with

$$\mathbf{\Omega}_{n+1} = \mathbf{\Omega}_n + h(\tilde{\omega}_n \mathbf{\Omega}_n) \tag{13.10}$$

If we're using a quaternion representation for orientation, we use a similar approach. We take our angular velocity vector and convert it to a quaternion $\mathbf{w}$, where

$$\mathbf{w} = (0, \omega)$$

We can multiply this by one-half of our original quaternion to get the derivative in quaternion form, giving us, again with Euler's method,

$$\mathbf{q}_{n+1} = \mathbf{q}_n + h\left(\frac{1}{2}\mathbf{w}_n\,\mathbf{q}_n\right) \tag{13.11}$$

A derivation of this equation is provided by Witken and Baraff [121] and Eberly [27], for those who are interested.

Using either of these methods allows us to integrate orientation. As far as updating angular velocity, computing acceleration for rotational dynamics is rather complicated, so we won't be using angular acceleration at all. Instead, since torque is the derivative of angular momentum, we'll integrate the torque to update angular momentum, and then compute the angular velocity from that. As when we integrated force, we'll need a function to compute total torque across the entire interval, called `CurrentTorque()`. For both methods, we'll have to modify our input variables to take into account orientation and angular velocity as well as position and velocity.

To find the angular velocity, we rewrite equation 13.8 to solve for ω:

$$\omega = \mathbf{J}^{-1}\mathbf{L} \tag{13.12}$$

When computing the angular velocity in this way, there is one detail that needs to be managed carefully. The inertia tensor is in the model space of the object. However, angular momentum is integrated from torque, which is computed in world space, and we want our resulting angular velocity to also be in world space. To keep things consistent, we need a way to convert our model space $\mathbf{J}^{-1}$ to world space. If we're using a rotation matrix to represent orientation, we can use it to transform $\mathbf{L}$ from world to model space, apply the inverse inertia tensor, and then transform back into world space. So, for a given time step,

$$\omega_{i+1} = \mathbf{\Omega}_{i+1}\mathbf{J}^{-1}\mathbf{\Omega}_{i+1}^{T}\mathbf{L}_{i+1} \tag{13.13}$$

If we're using quaternions, the most efficient way to handle this is to convert our quaternion to a matrix, and then compute equation 13.13.

Using semi-implicit Euler and quaternions, the full code for handling rotational quantities looks like the following.

```
// compute new angular momentum, orientation
mAngMomentum += h*CurrentTorque( mTranslate, mVelocity,
                                    mRotate, mAngVelocity);
mAngMomentum.Clean();

// update angular velocity
IvMatrix33 rotateMat(mRotate);
IvMatrix33 worldMomentsInverse =
           rotateMat*mMomentsInverse*::Transpose(rotateMat);
mAngVelocity = worldMomentsInverse*mAngMomentum;
mAngVelocity.Clean();
IvQuat w = IvQuat( 0.0f, mAngVelocity.x,
                   mAngVelocity.y, mAngVelocity.z );
mRotate += h*0.5f*w*mRotate;
mRotate.Normalize();
mRotate.Clean();
```

13.5 Collision Response

Up to this point, we haven't considered collisions. Our objects are moving gracefully through the world, speeding up or slowing down as we adjust our forces. All of which is accurately modeled, except that the objects go right through each other. Not a very realistic or fun game. Instead, we'll need a way to simulate the two objects bouncing away from each other due to the collision. We can do so by using the methods we've discussed in Chapter 12 in combination with some new techniques.

13.5.1 Contact Generation

For the purposes of this discussion, we'll assume a simple collision model, where the objects are convex and there is a single collision point. To perform our collision response properly, we have to know two things about the collision. The first is the point of contact between the two objects A and B—in other words, the point on the objects where they just touch (Figure 13.13). Since the two objects are just touching, there is a tangent plane that passes between the two, which also intersects both at that point. This is represented in the figure as a line. The second thing we need to know is the normal $\hat{\mathbf{n}}$ to that plane. We'll choose our normal to point from A, the first object; to B, the second.

Our main problem in figuring out collision location is that we're trying to detect collisions within an interval of time. In one time step, two objects may

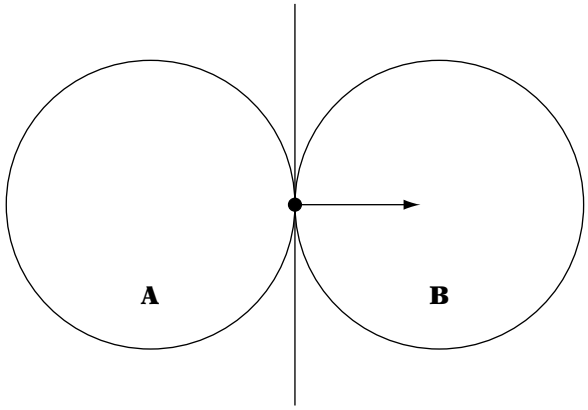


FIGURE 13.13 Point of collision. At the moment of impact between two convex objects, there is a single point of collision. Also shown is the collision plane and its normal.

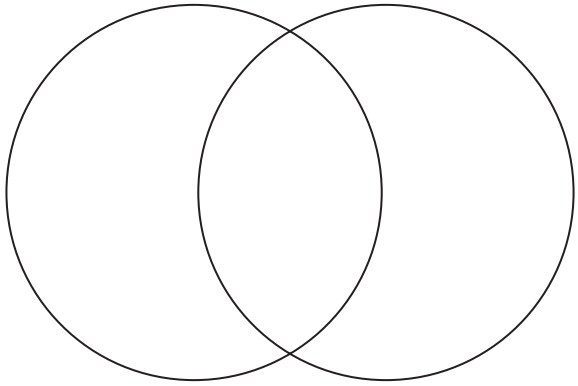


FIGURE 13.14 Interpenetrating objects. There is no single point of collision.

be completely separate; in the next, they are colliding. In fact, in most cases when collision is detected, we have missed the initial point of collision and the objects are already interpenetrating (Figure 13.14). Because of this, there is no single point of collision.

One possibility for finding the exact point when initial collision occurs is to do a binary search within the time interval. We begin by running our simulation and then testing for collisions. If we find one, and the two objects involved are interpenetrating, we step the entire simulation back half a time step and check again. If there is still penetration, we go back a quarter of the original time step, otherwise, we go forward a quarter of the original time

step. We keep doing this, ratcheting time forward or back by smaller and smaller intervals until we get an exact point of collision (unlikely) or we reach a certain level of iteration. At the end of the search, we'll either have found the exact collision point or will be reasonably close.

This technique has a few flaws. First of all, it's slow. Chances are that every time you get a collision, you'll need to run the simulation at least two or three additional times to get a point where the objects are just touching. In addition, in order for detection to be perfectly accurate, you need to rerun the simulation for all the objects, because their position at the time of the collision will be slightly different than their position at the end of the time interval. This may affect which objects are colliding. So, you need to run the simulation back, determine the collision point, apply the collision response, and then run the simulation forward until you hit another collision, do another binary search, and so on. In the worst case, with many colliding objects, your simulation will get bogged down, and you'll end up with long frame times. The accuracy of this method may be suitable for offline simulation, but it's not good for interactivity.

Another possibility is to ignore it, approximate the contact point and normal, and let the collision response push the two objects apart. This can work, but if the response is too slow, the two objects may remain interpenetrated for a while. This can look quite odd and may ruin the illusion of reality.

The third alternative begins by looking at the overlap between the two objects. The longest distance along that overlap is known as the penetration distance. We can push the two objects apart by the penetration distance until they just touch, and then use the point and normal from that intersection for collision calculations.

For example, take two spheres (Figure 13.15), with centers C_a and C_b and radii r_a and r_b. If we subtract one center C_a from the other center C_b, we get

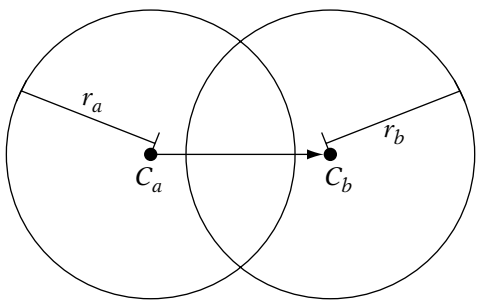


Figure 13.15 Determining penetration distance and collision normal.

the direction for our collision normal. The penetration distance p is then the sum of the two radii minus the length of this vector, or

$$p = (r_a + r_b) - \|C_b - C_a\| \tag{13.14}$$

We can move each sphere in opposite directions along this normal by the distance $p/2$, which will move them to a position where they just touch. This assumes that both objects can move — if one is not expected to move, like a boulder or a church, we translate the other object by the entire normal length. So, for two moving objects A and B, the formula is

```
mTranslate -= 0.5f*penetration*centerDiff;
other->mTranslate += 0.5f*penetration*centerDiff;
```

Once we've pushed them apart, the collision point is where our center difference vector crosses the boundary of the two spheres. We can compute this point by halving the difference vector and adding it to the old C_a. We finish up by normalizing the difference vector to get our collision normal.

Handling penetration distance for capsules is just as simple. Instead of using the center points to compute the collision normal, we use the closest points on the line segments that define each capsule. The penetration distance becomes the sum of the radii minus the distance between these points. For bounding boxes, Eberly [25] provides a method that computes the penetration distance between two oriented boxes.

This technique does have some flaws. First, pushing the two objects apart by the entire penetration distance may look too abrupt. Instead, we can push them apart by a fraction of the penetration distance and assume that the collision response will separate them the rest of the way. The slight interpenetration will only be noticeable for one or two frames. Second, if objects are moving fast enough and the collision is detected too late, the two objects may pass through each other. If this case is not handled in the collision detection, we will get some very odd results when the objects are pushed apart. Finally, because we're pushing objects away from each other instantaneously, we may end up with situations where two objects collide, and one of them is moved into a third, causing a new interpenetration. Because we may have already tested for collision between the second pair of objects, we'll miss this collision. If we're expecting a large number of collisions between close objects, this simple system may not be practical.

As a final note on contact generation, usually the collision-detection system will generate a pair of contact features, one for each object, per collision. There may be multiple contacts per object (think of a book resting on its edge, or even its face), and there may be dependencies between many objects

that control how contacts are resolved (think of a stack of boxes). We'll briefly discuss how to manage such problems later, but for our main thread of discussion we'll concentrate on single points of contact.

13.5.2 Linear Collision Response

Source Code Demo LinCollision

Whatever method we use, we now have two of the properties of the collision we need to compute the linear part of our collision response: a collision normal $\hat{\mathbf{n}}$ and a collision point P. The other two elements are the incoming velocities of the two objects, $\mathbf{v}_a$ and $\mathbf{v}_b$. Using this information, we are finally ready to compute our collision response.

The technique we'll use is known as an *impulse-based* system. The idea is that near the time of collision, the forces and position remain nearly constant, but there is a discontinuity in the velocity. At one point in time, the velocities of the objects are heading toward one another; in the next infinitesimal moment later, they are heading away. How much and in what relation the velocities change depends on the magnitude and direction of the incoming velocities, the direction of the collision normal, and the masses of the two objects.

Let's look again at the simple case of our two spheres A and B (Figure 13.16(a)). For now, let's assume their masses are equal. We again see our two incoming velocities $\mathbf{v}_a$ and $\mathbf{v}_b$ and our collision normal $\hat{\mathbf{n}}$. The idea is that we want to modify our velocity by an *impulse* that is normal to the point of collision. The impulse will act to push the two objects apart—if the masses are equal, it will be equal in magnitude, but opposite in direction for each object. So, we need to generate a scale factor j for our collision normal, and then add the scaled collision normal $j\hat{\mathbf{n}}$ and $-j\hat{\mathbf{n}}$ to $\mathbf{v}_a$ and $\mathbf{v}_b$ to get our outgoing velocities. So, in order to compute the impulse vector, we need to compute this factor j.

To begin our computation,we need the relative velocity $\mathbf{v}_{ab}$, which is just $\mathbf{v}_a - \mathbf{v}_b$ (Figure 13.16(a)). From that, we'll compute the amount of relative velocity that is applied along the collision normal (Figure 13.16(b)). Recall that the dot product of any vector with a normalized vector gives the projection along the normal vector, which is just what we want. So,

$$\mathbf{v}_n = (\mathbf{v}_{ab} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

At this point, we do one more test to see if we actually need to calculate an impulse vector. If the relative velocity along the collision normal is negative, then the two objects are heading away from each other and we don't need to compute an impulse. We can break out of the collision-response code and proceed to the next collision. Otherwise, we continue with computing j.

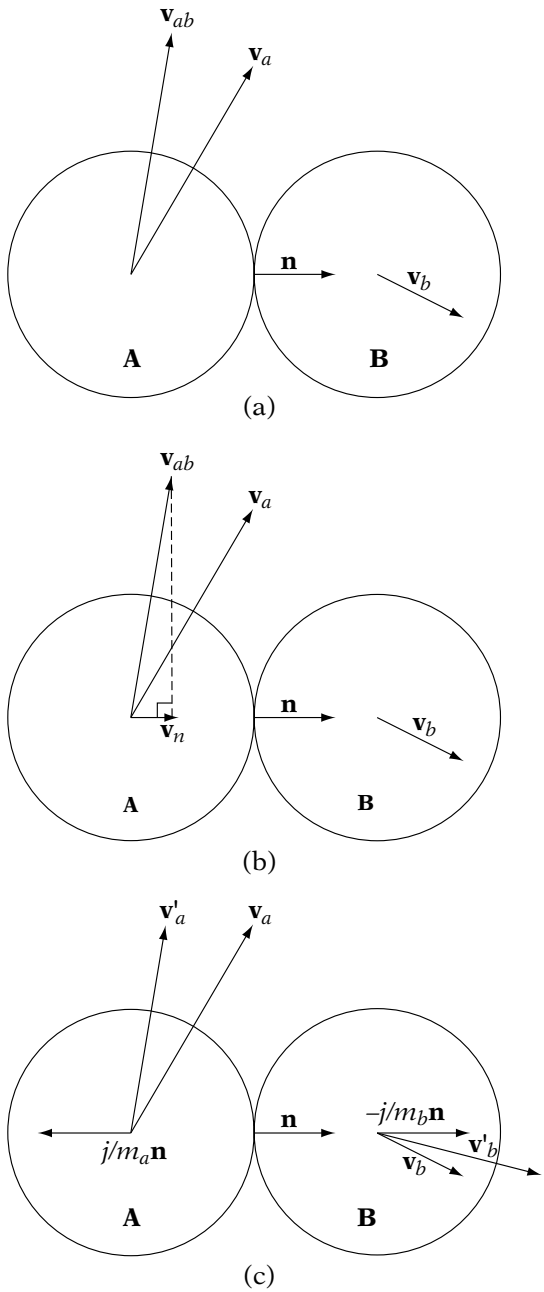


FIGURE 13.16 (a) Computing collision response. Calculating relative velocity. (b) Collision response. Computing relative velocity along normal. (c) Collision response. Adding impulses to create outgoing velocities.

In order to compute a proper impulse, two conditions need to be met. First of all, we need to set the ratio of the outgoing velocity along the collision normal to the incoming velocity. We do this by setting a *coefficient of restitution* ϵ:

$$\mathbf{v}'_n = -\epsilon \mathbf{v}_n$$

or

$$(\mathbf{v}'_a - \mathbf{v}'_b) \cdot \hat{\mathbf{n}} = -\epsilon(\mathbf{v}_a - \mathbf{v}_b) \cdot \hat{\mathbf{n}} \tag{13.15}$$

Each object will have its own value of ϵ. This simulates two different physical properties. First of all, when one object collides with another some energy is lost, usually in the form of heat. Second, if the object is somewhat soft and/or sticky, or *inelastic*, the bonding forces between it and its target will decrease the outgoing velocities. Elastic in this case doesn't refer to the stretchiness of the object, but how resilient it is. A superball is not very malleable, but has very elastic collisions. So, the quantity ϵ represents how much energy is lost and how elastic the collision between the two objects is. If both objects have an ϵ of 1, then they will bounce away from each other with the same relative velocity they had coming in. If both objects have an ϵ of 0, they will stick together like two clay balls and move as one. Values in between will give a linear range of elastic responsiveness. Values greater than 1 or less than 0 are not permitted. An ϵ greater than 1 would add energy into the system, so a ball bouncing on a flat surface would bounce progressively higher and higher. An ϵ less than 0 means that the objects would be highly attracted to each other upon collision and would lead to undesirable interpenetrations.

Even if energy is not quite conserved (technically it is, but we're not tracking the heat loss), then momentum is. Because of this, the total momentum of the system of objects before and after the collision needs to be equal. So,

$$m_a \mathbf{v}_a + j\hat{\mathbf{n}} = m_a \mathbf{v}'_a$$

or

$$\mathbf{v}'_a = \mathbf{v}_a + \frac{j}{m_a}\hat{\mathbf{n}} \tag{13.16}$$

Similarly,

$$m_b \mathbf{v}_b - j\hat{\mathbf{n}} = m_b \mathbf{v}'_b$$

or

$$\mathbf{v}'_b = \mathbf{v}_b - \frac{j}{m_b}\hat{\mathbf{n}} \tag{13.17}$$

With this, we finally have all the pieces that we need. If we substitute equations 13.16 and 13.17 into equation 13.15 and solve for j, we get the final impulse factor equation:

$$j_a = \frac{-(1+\epsilon_a)\,\mathbf{v}_{ab} \cdot \hat{\mathbf{n}}}{\left(\frac{1}{m_a} + \frac{1}{m_b}\right)} \tag{13.18}$$

The equation for j_b is similar, except that we substitute ϵ_b for ϵ_a.

Now that we have our impulse values, we substitute them back into equations 13.16 and 13.17, respectively, to get our outgoing velocities (Figure 13.16(c)). Note the effect of mass on the outgoing velocities. As we expect, as the mass of an object grows larger, it grows more resistant to changing its velocity due to an incoming object. This is counteracted by j, which grows as relative velocity increases, or as the combined masses increase.

Our final algorithm for collision response between two spheres is as follows.

```
float radiusSum = mRadius + other->mRadius;
collisionNormal = other->mTranslate - mTranslate;
float distancesq = collisionNormal.LengthSquared();
// if distance squared < sum of radii squared, collision!
if ( distancesq <= radiusSum*radiusSum )
{
     // handle collision
     // penetration is distance - radii
     float distance = ::IvSqrt(distancesq);
     penetration = radiusSum - distance;
     collisionNormal.Normalize();

     // collision point is average of penetration
     collisionPoint = 0.5f*(mTranslate + mRadius*collisionNormal)
                 + 0.5f*(other->mTranslate - other->mRadius*collisionNormal);

     // push out by penetration
     mTranslate -= 0.5f*penetration*collisionNormal;
     other->mTranslate += 0.5f*penetration*collisionNormal;

     // compute relative velocity
     IvVector3 relativeVelocity = mVelocity - other->mVelocity;

     float vDotN = relativeVelocity*collisionNormal;
     if (vDotN < 0)
          return;
```

```
        // compute impulse factor
        float modifiedVel = vDotN/(1.0f/mMass + 1.0f/other->mMass);
        float j1 = -(1.0f+mElasticity)*modifiedVel;

        float j2 = -(1.0f+other->mElasticity)*modifiedVel;

        // update velocities
        mVelocity += j1/mMass*collisionNormal;
        other->mVelocity -= j2/other->mMass*collisionNormal;
    }
```

In this simple example, we have interleaved the sphere collision detection with the computation of the collision point and normal. This is for efficiency's sake, since both use the sum of the two radii and the difference vector between the two centers for their computations. As mentioned above, a more complex collision system will generate contact pairs to be fed to the collision-response system.

13.5.3 Rotational Collision Response

This is all well and good, but most objects are not spheres, which means that they have a visible orientation. When one collides with another at an offset to the center of mass, we would expect some change in angular velocity as well as linear velocity. In addition, any incoming angular velocity should affect the collision as well. A cue ball with spin (or English) applied causes a much different effect on a target pool ball than a cue ball with no spin.

As with linear and rotational dynamics, the way we handle rotational collision response is very similar to how we handle linear collision response. We need to modify only a few equations and recalculate our impulse factor j.

One modification we have to make is the effect of angular velocity on the incoming velocity. Up to this point, we've assumed that when the two objects strike each other, their surfaces are not moving, so the velocity at the collision point is simply the linear velocity. However, if one or both of the objects are rotating, then there is an additional velocity factor applied at the point of collision, as one surface passes by the other. Recall that equation 13.6 allows us to take an angular velocity ω and a displacement from the center of mass $\mathbf{r}$ and compute the linear velocity contributed by the angular velocity at the point of displacement. Adding this to the original incoming velocities, we get

$$\bar{\mathbf{v}}_a = \mathbf{v}_a + \omega_a \times \mathbf{r}_a$$

$$\bar{\mathbf{v}}_b = \mathbf{v}_b + \omega_b \times \mathbf{r}_b$$

Now the relative velocity $\mathbf{v}_{ab}$ at the collision point becomes

$$\mathbf{v}_{ab} = \bar{\mathbf{v}}_a - \bar{\mathbf{v}}_b$$

and equation 13.15 becomes

$$(\bar{\mathbf{v}}'_a - \bar{\mathbf{v}}'_b) = -\epsilon(\bar{\mathbf{v}}_a - \bar{\mathbf{v}}_b) \tag{13.19}$$

The other change needed is that in addition to handling linear momentum, we also need to conserve angular momentum. This is a bit more complex compared to the equations for linear motion, but the general concept is the same. The outgoing angular momentum should equal the sum of the incoming angular momentum and any momentum imparted by the collision. For object A, this is represented by

$$\mathbf{I}_a\omega_a + \mathbf{r}_a \times j\hat{\mathbf{n}} = \mathbf{I}_a\omega'_a \tag{13.20}$$

or

$$\omega'_a = \omega_a + \mathbf{I}_a^{-1}(\mathbf{r}_a \times j\hat{\mathbf{n}}) \tag{13.21}$$

For object B, this is

$$\mathbf{I}_b\omega_b - \mathbf{r}_b \times j\hat{\mathbf{n}} = \mathbf{I}_b\omega'_b \tag{13.22}$$

or

$$\omega'_b = \omega_b - \mathbf{I}_a^{-1}(\mathbf{r}_b \times j\hat{\mathbf{n}}) \tag{13.23}$$

Just as with linear collision response, we can substitute equations 13.21 and 13.23 into 13.19, and together with equations 13.16 and 13.17, solve for j to get

$$j = \frac{-(1+\epsilon)\,\mathbf{v}_{ab} \cdot \hat{\mathbf{n}}}{\left(\frac{1}{m_a} + \frac{1}{m_b}\right) + \left[(\mathbf{I}_a^{-1}(\mathbf{r}_a \times \hat{\mathbf{n}})) \times \mathbf{r}_a + (\mathbf{I}_b^{-1}(\mathbf{r}_b \times \hat{\mathbf{n}})) \times \mathbf{r}_b\right] \cdot \hat{\mathbf{n}}} \tag{13.24}$$

Using this modified j value we calculate new angular momenta using equations 13.20 and 13.22 and from that calculate angular velocity as we did with angular dynamics, using equation 13.8. We use this same j for our linear collision response as well. And of course, as before we'll use different ϵs for the two objects.

We change our linear collision–handling code in three places to achieve this. First of all, the relative velocity collision incorporates incoming angular velocity, as follows.

```
    // compute relative velocity
    IvVector3 r1 = collisionPoint - mTranslate;
    IvVector3 r2 = collisionPoint - other->mTranslate;
    IvVector3 vel1 = mVelocity + Cross( mAngularVelocity, r1 );
    IvVector3 vel2 = other->mVelocity + Cross( other->mAngularVelocity,
r2 );
    IvVector3 relativeVelocity = vel1 - vel2;
```

Then, we add angular factors to our calculation for j, as follows.

```
    // compute impulse factor
    float denominator = (1.0f/mMass
         + 1.0f/other->mMass)*(collisionNormal.Dot(collisionNormal));

    // compute angular factors
    IvVector3 cross1 = Cross(r1, collisionNormal);
    IvVector3 cross2 = Cross(r2, collisionNormal);
    cross1 = mWorldMomentsInverse*cross1;
    cross2 = other->mWorldMomentsInverse*cross2;
    IvVector3 sum = Cross(cross1, r1) + Cross(cross2, r2);
    denominator += (sum.Dot(collisionNormal));
    float modifiedVel = vDotN/denominator;
```

Finally, in addition to linear velocity, we recalculate angular velocity, as follows.

```
// update angular velocities
mAngularMomentum += Cross(r1, j1*collisionNormal);
mAngularVelocity = mWorldMomentsInverse*mAngularMomentum;
other->mAngularMomentum += Cross(r2, j2*collisionNormal);
other->mAngularVelocity = mWorldMomentsInverse*other->mAngularMomentum;
```

13.5.4 Extending the System

Everything up to this point will provide a reasonable rigid-body simulation, with moving and colliding bodies. However, there may be some additional

features we may want to add. The following present some possible solutions for expanding and extending our simple system.

Resting Contact

The methods we described above handle the case when two objects are heading toward each other along the collision normal. Obviously, if they're heading apart, we don't need to consider these methods — they are separating. However, if their relative velocity along the normal is 0, then we have what is called a *resting contact*. A simple example of a resting contact is a box sitting on the floor; it has no downward velocity and yet it is in contact with the floor.

While in general we wouldn't expect that we would have to handle a resting contact, consider the case when the box is being affected by gravity. After one time step it will have a downward velocity into the floor, and then we'll have to handle it as a colliding contact. However, doing so will lead to the box leaping up into the air as we subtract out the initial velocity and then add the response due to the impulse. The box will fall again due to gravity, and then bounce up, and we'll get a very jittery result. Obviously we'd like to deal with the resting contact before this occurs.

One solution is to compute a force that counteracts the force of gravity. This is known as a *constraint force*, as we're constraining the box from passing through the floor. This is certainly a reasonable solution in the absence of other forces, but suppose we now have two boxes stacked on top of one another. We'll need some way to transfer that constraint force up to the next box to make sure they both don't move, in addition to preventing interpenetration between the boxes. When using constraint forces, things can get very complicated very fast.

A less accurate but more tractable alternative is to use a modification of our impulse method. This is known as a *micro-impulse engine*, as our impulses due to resting contact will be very small. The key to a micro-impulse engine is to add the right amount of correction to ensure that objects don't pass through each other and don't bounce. Millington [78] detects the case that we described above by comparing the velocity generated from the current frame to the object's current velocity. If it's less, then we continue with normal collision resolution, otherwise we know it's the resting case. Catto [17] does something similar, but uses an iterative process to lower the impulse value (see below). In either case, it requires only minor tweaks to our basic algorithm to get some very nice results.

Constraints

As mentioned, resting contact can be thought of as a constraint on our system, as it is preventing us from pushing an object through a surface. There are other

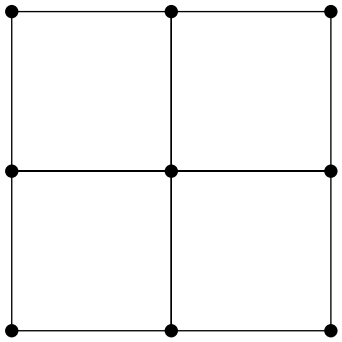


FIGURE 13.17 Mesh of particles constrained by distance.

constraints we can set up similarly. For example, suppose we have a collection of particles, and we want to keep each of them a fixed distance away from their neighbors, say in a grid (Figure 13.17). This is particularly useful when trying to simulate cloth. We can also have joint contraints, which keep two points coincident while allowing the remainder of the objects to swing free. And the list goes on. Any case that describes a fixed relationship between two objects can be modeled as a constraint.

Constraints are particularly useful in modeling a class of objects known as *soft bodies*. We've already mentioned cloth, above. Similar principles can be applied to simulate rope. When we build a simple hierarchical system, we get a skeleton that can be used to simulate a dead or unconscious figure, known as rag-doll physics. Therefore, contraints are extremely powerful in creating a new sort of interaction in our world.

We could implement these constraints as springs, but as we've seen, stiff springs cause us a lot of problems when integrating. An alternative is to compute the exact force to keep the two objects constrained, as was suggested with resting contact. However, as before, with multiple objects this can get quite complex and requires yet another system to be added to our simulation engine.

Fortunately, impulses can work in this case, too. As mentioned, collision and resting contact are just two kinds of constraint. To model others, we just need to compute the necessary impulse to keep the two objects from breaking the constraint condition and no more. This has the noted advantage that it works well with our existing impulse system for collisions and resting contacts. It's also usually simpler to compute an impulse that keeps two objects constrained than a force, as we're removing one level of indirection from position and orientation.

For those interested, details for building various types of constraint systems can be found in Catto [121], Jakobson [61], Millington [78], and Witkin and Baraff [16].

Multiple Points

The final issue we'll discuss is how to manage multiple constraints and contacts, both on one object and across multiple objects. In reality, our constraint forces and contact impulses are occurring simultaneously so the most accurate way to handle this is to build a large system of equations and solve for them all at once. This is usually a quite complex process, both in constructing the equations and solving them. While it often ends up as a linear system, using Gaussian elimination is too expensive due to the large numbers of equations involved. Instead, an iterative process such as the Gauss-Seidel or Jacobi method is used. In principle, this is similar to Newton's method in that it involves computing an initial approximation and then refining that approximation to converge on the final answer.

An alternative, suggested in different ways by Catto [17] and Millington [78], is to continue to update impulses sequentially. However, instead of updating once per contact pair, we take a page from the iterative methods and update each pair as necessary, until a certain level of convergence is reached. Millington's method is to iterate through the contact pairs, finding the ones with the deepest penetration and resolving them first. One set of pairs may be revisited because it is affected by one or more other sets of pairs. In this way the impulses are iteratively adjusted until hopefully they converge on a reasonable solution.

Catto's method, on the other hand, involves updating the impulse values at each contact pair for several iterations, then applying the impulses when done. This has the advantage that it can cut down on jitter. Normally, impulses are required to be positive, so what happens is that any correction in the negative direction will be clamped to zero. This means that we can get overcorrection where objects bounce into the air briefly and then settle back down much as we saw with resting contact. Instead, Catto recommends accumulating the impulse value, including the incorporation of negative values. He has also found that doing this while clamping the accumulated impulse is equivalent to an iterative matrix method known as *projected Gauss-Seidel*, which is a common variant used for solving constraint systems. This provides an excellent mathematical justification for this approach.

As before, details on solving these issues can be found in Catto [17], Jakobson [61], Millington [78], and Witkin and Baraff [121]. Golub and Van Loan [44] have information on Gauss-Seidel and Jacobi methods.

13.6 Efficiency

Now that we have a simple simulation system, some notes on using it efficiently may be appropriate. The first rule is that this is a game. Don't waste

time with any more processing power than you need to get the effect you want. While a fully realistic simulation may be desirable, it can't take too much processing power away from the other subsystems, for instance, graphics or artificial intelligence. How resources are allocated among subsystems in a game depends on the game's focus. If a simpler solution will come close enough to the appearance of realism, then it is sometimes better to use that instead.

One way to reduce the amount of resources used is to simplify the problem. So far we've been assuming that we're building a truly 3D game, where the objects need to move in three degrees of freedom. If, however, you were building a tank game, it's highly unlikely that the tank would leave the ground. In most cases, land warfare games take place on a 2D map, with some height variation, so with the exception of projectiles the entire situation is really a 2D problem. You don't have to consider gravity, as angular dynamics is constrained to just rotation around z, and thus you really need only one factor for your moments of inertia. This considerably simplifies the angular dynamics equations. The same is true for a first-person shooter; in general, characters will interact as cylinders sliding on a flat floor, with vertical walls as boundaries. In this case, we can simplify the collision problem to circles on a 2D plane.

Another way to improve efficiency is to run simulation code only on some of the objects in the world. For example, we could restrict full simulation to those objects that are visible or near the player. We could use a simplified simulation model for the other objects or not move them at all. We could also not simulate objects that aren't currently moving, and begin simulation only when forces are applied or another object collides with them. When using this technique, we need to be careful about discontinuities in the simulation. We don't want a falling object that passes out of view to stop in midair, only to start falling again when it's visible again. Nor do we want objects to jerk, move strangely, or jump position as one simulation model ceases and another takes over. While managing these discontinuities can be tricky, using such restrictions can also gain quite a performance boost.

Simplifying the forces computed during simulation is another place to find speed improvements. We've alluded to this before. In a truly complete simulation we would compute a gravitational force, a normal force to keep the object from sinking through the ground, and a static frictional force to keep the object from sliding down any inclines. In most cases, we can assume that the sum of all these forces is zero and ignore them completely. We really haven't covered friction in any detail, but it's a similar case. We could compute a complex equation for an object that handles all contact points, current surface area, and whether we are moving or at rest, or we could just use a drag coefficient multiplied by velocity. If your game calls for the full friction model, then by all means do it, but in many cases, it can be overkill.

13.7 Chapter Summary

The use of physical simulation is becoming an important part of providing realistic motion in games and other interactive applications. In this chapter we have described a simple physical simulation system, using basic Newtonian physics. We covered some techniques of numeric integration, starting with Euler's method, and discussed their pros and cons. Using these integration techniques, we have created a simple system for linear and rotational rigid-body dynamics. Finally, we have shown how we can use the results of our collision system to generate impulses for collision response.

The system we've presented is a very simple one — we've barely scratched the surface of what is possible in terms of physical simulation. For those who are interested in proceeding further, Eberly [27] presents a more complete look at the mathematics in game physics, including the use of physics in graphics shaders. Millington [78] presents the gradual development of a simple physics engine that is suitable for game engines. Burden and Faires [14] and Golub and Ortega [43] have more descriptions of numerical integration techniques and managing error bounds. Finally, Witken and Baraff [121], Jakobson [61], and Catto [16] describe different methods for building constraint systems.

This page intentionally left blank

Bibliography

[1] AMD. AMD developer support website. *http://www.amd.com*.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std. 754-1985,* New York, 1985.

[3] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version*, 7th edition. John Wiley and Sons, New York, 1994.

[4] Sheldon Axler. *Linear Algebra Done Right*, 2nd edition. Springer-Verlag, New York, 1997.

[5] Martin Baker and Michael Norel. Euclideanspace website. *http://www.euclideanspace.com*.

[6] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann Publishers, San Francisco, 1987.

[7] J. F. Blinn and M. E. Newell. Clipping using homogeneous coordinates. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 245–251, ACM, New York, 1978.

[8] Jim Blinn. *A Trip Down the Graphics Pipeline*. Morgan Kaufmann Publishers, San Francisco, 1996.

[9] Jim Blinn. *Notation, Notation, Notation*. Morgan Kaufmann Publishers, San Francisco, 2002.

[10] Jonathan Blow. Hacking quaternions. *Game Developer*, March 2002.

[11] Jonathan Blow and Atman J. Binstock. How to find the inertia tensor (or other mass properties) of a 3D solid body represented by a triangle mesh. Technical report, *http://number-none.com*, 2004.

[12] W. Boehm. Inserting new knots into b-spline curves. *Computer Aided Design*, 12(4):199–201, 1980.

[13] W. Boehm. On cubics: A survey. *Computer Graphics and Image Processing*, 19:201–226, 1982.

[14] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*, 5th edition. PWS Publishing Company, Boston, 1993.

[15] Thomas Busser. Polyslerp: A fast and accurate polynomial approximation of spherical linear interpolation (slerp). *Game Developer*, February 2004.

[16] Erin Catto. Iterative dynamics with temporal coherence. Technical report, Crystal Dynamics, 2005.

[17] Erin Catto. Fast and simple physics using sequential impulses. GDC 2006 Tutorial: Physics for Game Programmers, 2006.

[18] Arthur Cayley. *The Collected Mathematical Papers of Arthur Cayley*. Cambridge University Press, Cambridge, 1889–1897.

[19] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers, San Francisco, 1993.

[20] T. N. Cornsweet. *Visual Perception*. Academic Press, New York, 1970.

[21] R. Courant and D. Hilbert. *Methods of Mathematical Physics, Volume One*. John Wiley and Sons, New York, 1989 (reprint).

[22] M. Cyrus and J. Beck. Generalized two- and three-dimensional clipping. *Computers and Graphics*, 3:23–28, 1978.

[23] Tony deRose. Three-dimensional computer graphics: A coordinate-free approach. Technical report, University of Washington, 1993.

[24] Rene Descartes. *La geometrie (The Geometry of Rene Descartes)*. Dover Publications, New York, 1954.

[25] David H. Eberly. *3D Game Engine Design*. Morgan Kaufmann Publishers, San Francisco, 2001.

[26] David H. Eberly. Rotation representations and performance issues. Technical report, Geometric Tools, Inc., 2002.

[27] David H. Eberly. *Game Physics*. Morgan Kaufmann Publishers, San Francisco, 2003.

[28] David H. Eberly. Eigensystems for 3×3 symmetric matrices (revisited). Technical report, Geometric Tools Inc., 2006.

[29] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texture and Modelling: A Procedural Approach*, 3rd edition. Morgan Kaufmann Publishers, San Francisco, 2003.

[30] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[31] Wolfgang Engel, editor. *Direct3D ShaderX*. Wordware, Plano, TX, 2002.

[32] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Publishers, San Francisco, 2004.

[33] Euclid. *The Elements*. Dover Publications, New York, 1956.

[34] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA, 2001.

[35] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, Reading, MA, February 2003.

[36] Randima Fernando, editor. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, Reading, MA, March 2004.

[37] Agner Fog. Pseudo random number generators. *http://www.agner.org/random*.

[38] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*, 2nd edition. Addison-Wesley, Reading, MA, 1992.

[39] Stephen H. Friedberg, Arnold J. Insel, and Lawrence E. Spence. *Linear Algebra*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[40] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, Boston, 1989.

[41] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, San Francisco, 1994.

[42] Ronald N. Goldman. Decomposing linear and affine transformations. In David Kirk, editor, *Graphics Gems III*, pages 108–116. Academic Press, San Diego, 1992.

[43] Gene H. Golub and James M. Ortega. *Scientific Computing and Differential Equations: An Introduction to Numerical Methods*. Academic Press, Boston, 1992.

[44] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1993.

[45] Larry Gonick and Woollcott Smith. *The Cartoon Guide to Statistics*. Harper Collins, New York, 1993.

[46] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 171–180, 1996.

[47] Jens Gravesen. The length of bezier curves. In Alan Paeth, editor, *Graphics Gems V*, pages 199–205. Academic Press, Boston, 1998.

[48] Kris Gray. *Microsoft DirectX9 Programmable Graphics Pipeline*. Microsoft Press, Redmond, WA, 2003.

[49] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, Providence, R.I., 2003.

[50] Brian Guenter and Richard Parent. Computing the arc length of parametric curves. *IEEE Computer Graphics and Applications*, 10(3):72–78, 1990.

[51] Philippe Guigue and Olivier Devillers. Fast and robust triangle-triangle overlap using orientation predicates. *Journal of Graphics Tools*, 8(1):25–32, 2003.

[52] William Hamilton. On quaternions, or on a new system of imaginaries in algebra. *Philosophical Magazine*, 1844–1850 (available online).

[53] A. Hanson and H. Ma. Parallel transport approach to curve framing. Technical report 425, Indiana University Computer Science Department, 1995.

[54] Donald Hearn and M. Pauline Baker. *Computer Graphics*, 2nd edition. Prentice-Hall, Upper Saddle River, NJ, 1996.

[55] Paul Heckbert. Texture mapping polygons in perspective. Technical report, New Institute of Technology, 1983.

[56] Paul Heckbert and Henry Moreton. Interpolation for polygon texture mapping and shading. In David Rogers and Rae Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, New York, 1991.

[57] Chris Hecker. Under the hood/behind the screen: Perspective texture mapping (series). *Game Developer Magazine*, 1995–1996.

[58] Martin Held. Erit—A collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[59] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd edition. Morgan Kaufmann Publishers, San Francisco, 1996.

[60] Intel. Intel developer support website. *http://developer.intel.com*.

[61] Thomas Jakobson. Advanced character physics. In *Proceedings of Game Developers Conference*, 2001.

[62] William Kahan. Lecture notes on the status of IEEE-754. Postscript file accessible electronically through the Internet at *http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps*, 1996.

[63] Michael Kallay. Computing the moment of inertia of a solid defined by a triangle mesh. *Journal of Graphics Tools*, 11(2):51–57, 2006.

[64] Khronos Group Developers' Website. *http://www.khronos.org*.

[65] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, 3rd edition. Addison-Wesley, Reading, MA, 1993.

[66] Doris H. U. Kochanek and Richard H. Bartels. Interpolating splines with local tension, continuity, and bias control. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 33–41, 1984.

[67] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[68] Yu-Dong Liang and Brian Barsky. A new concept and method for line clipping. *ACM Transactions on Graphics*, 3(1):1–22, 1984.

[69] Marta Löfstedt and Tomas Akenine-Möller. An evaluation framework for ray-triangle intersection algorithms. *Journal of Graphics Tools*, 10(2): 13–26, 2005.

[70] D. Malacara. *Color Vision and Colorimetry: Theory and Applications*. SPIE Press, Bellingham, WA, 2002.

[71] George Marsaglia. Random numbers fall mainly in the planes. In *Proceedings of the National Academy of Sciences*, 61:25–28, 1968.

[72] George Marsaglia. Remarks on choosing and implementing random number generators. *Communications of the ACM*, 36(7):105–108, 1993.

[73] George Marsaglia. Yet another RNG, Posting in *sci.stat.math*, August 1, 1994.

[74] George Marsaglia and Arif Zaman. A new class of random number generators. *The Annals of Applied Probability*, 1(3):462–480, 1991.

[75] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators. *ACM Transactions on Modelling and Computer Simulation*, 2:179–194, 1992.

[76] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator, *ACM Transactions on Modelling and Computer Simulation*, 8:3–30, 1998.

[77] Microsoft. Direct X SDK. Available for free download from *http://msdn.microsoft.com*.

[78] Ian Millington. *Game Physics Engine Development*. Morgan Kaufmann Publishers, San Francisco, 2006.

[79] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2):31–50, 1996.

[80] Tomas Möller. A fast triangle–triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[81] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[82] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*, 2nd edition. A. K. Peters, Natick, MA, 2002.

[83] Hubert Nguyen. Casting shadows. *Game Developer Magazine*, March 1999.

[84] NVIDIA. NVIDIA developer support website. *http://developer.nvidia.com*.

[85] OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, 3rd edition. Addison-Wesley, Reading, MA, 1999.

[86] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, 2000.

[87] Lewis Padgett. Mimsy were the borogroves. In Robert Silverberg, editor, *Science Fiction Hall of Fame, Volume 1*, Tor Books, New York, 2003.

[88] Rick Parent. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann Publishers, San Francisco, 2002.

[89] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[90] Ken Perlin. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 287–296, 1985.

[91] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers, San Francisco, 2004.

[92] Matt Pharr, editor. *GPU Gems 2: Mapping Computational Concepts to GPUs*. Addison-Wesley, Reading, MA, March 2005.

[93] Bui Tuong Phong. Illumination for computer-generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[94] Charles Poynton. Charles Poynton's color FAQ. *http://www.poynton.com/*.

[95] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1991.

[96] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edition. Cambridge University Press, New York, 1993.

[97] David F. Rogers. *An Introduction to NURBS: With Historical Perspective*. Morgan Kaufmann Publishers, San Francisco, 2000.

[98] David F. Rogers and J. Alan Adams. *Mathematical Elements for Computer Graphics*, 2nd edition. McGraw-Hill Inc., New York, 1990.

[99] Randi Rost. *OpenGL(R) Shading Language*. Addison-Wesley, Reading, MA, February 2004.

[100] Philip J. Schneider and David H. Eberly. *Geometric Tools for Computer Graphics*. Morgan Kaufmann Publishers, San Francisco, 2002.

[101] I. Schrage. A more portable fortran random number generator. *ACM Transactions of Mathematics Software*, 5(2):132–138, 1979.

[102] Hao Shen, Phen Ann Heng, and Zesheng Tang. A fast triangle-triangle overlap test using signed distances. *Journal of Graphics Tools*, 8(1):17–24, 2003.

[103] Ken Shoemake. Animating rotation with quaternion curves. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 245–254, 1985.

[104] Ken Shoemake. Quaternion calculus for animation. In *Math for SIGGRAPH (ACM SIGGRAPH '89 Course Notes 23)*, pages 187–205, 1989.

[105] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface '92*, pages 258–264, 1992.

[106] William Stallings. *Computer Organization and Architecture*, 5th edition. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[107] Dan Sunday. Distance between lines and segments with their closest point of approach. Technical report, *http://geometryalgorithms.com*, 2001.

[108] I. E. Sutherland. Sketchpad: A man-machine graphical communications system. In *IFIPS Proceedings of the Spring Joint Computer Conference*, 1963.

[109] I. E. Sutherland and G. W. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[110] Steve Theodore. Why be normal? *Game Developer Magazine*, October 2004.

[111] Andy Thomason. Faster quaternion interpolation using approximations. In Kim Pallister, editor, *Game Programming Gems 5*. Charles River Media, Hingham, MA, 2005.

[112] Gino van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers, San Francisco, 2003.

[113] James M. Van Verth. Using the covariance matrix for better-fitting bounding objects. In Andrew Kirmse, editor, *Game Programming Gems 4*. Charles River Media, Hingham, MA, 2004.

[114] James M. Van Verth. Spline-based time control for animation. In Kim Pallister, editor, *Game Programming Gems 5*. Charles River Media, Hingham, MA, 2005.

[115] Stephen Vincent and David Forsey. Fast and accurate parametric curve length computation. *Journal of Graphics Tools*, 6(4):29–40, 2001.

[116] David R. Warn. Lighting controls for synthetic images. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, 1983.

[117] Alan Watt and Fabio Policarpo. *3D Games: Real-Time Rendering and Software Technology, Volume One*. Addison-Wesley, Harlow, U.K., 2001.

[118] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *Lecture Notes in Computer Science, New Results and New Trends in Computer Science*, volume 555, pages 359–370. Springer-Verlag, New York, 1991.

[119] Amy Williams, Steve Barnes, H. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*, 10(1):49–54, 2005.

[120] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, 1983.

[121] Andrew Witkin and David Baraff. *Physically based modelling: Principles and practice*, SIGGRAPH 2001 Course Notes, 2001.

[122] George Wohlberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA, 1990.

INDEX[1]

A

AABBs, *see* Axis-aligned bounding boxes (AABBs)
Absolute error
  IEEE 754 floating-point standard, 12
  real number representation, 3
Acceleration, in linear dynamics
  constant acceleration, 602–605
  variable acceleration, 607–609
Acceleration vector, definition, 37
Addition
  catastrophic cancelation, 22
  fixed-point numbers, C-9
  floating-point addition, 22
  IEEE 754 floating-point standard, 13–14
  matrix addition, 90, 101
  modulate with late addition, 360
  quaternions, 187
  vectors, 37–40, 43
Add mode, pixel blending, 418
Adjoint matrix, and determinants, 128–129
Affine combination, and points, 68–69
Affinely independent points, definition, 69
Affine in screen space, definition, 384
Affine spaces
  components, 66–68
  definition, 66
Affine transformations
  around arbitrary point, 156–158
  definition, 133
  exercises, X-6
  formal definition, 136–137
  formal representation, 138–139
  game object manipulation
    model and world coordinate frames, 160–161
    overview, 159–160
    placement, 161–164
  matrix decomposition, 164–166
  matrix decomposition avoidance, 166–169
  matrix definition, 134–136
  plane normals, 158–159
  reflection, 151–154
  rotation, 141–150
  scaling, 150–151
  shear, 154–156
  standard types, 139
  translation, 139–141
AI agent, *see* Artificial intelligence (AI) agent
Algebraic rules
  dot product, 48
  scalar multiplication, 38–39
  vectors, 37–38
Algorithms, *see also* Sample code
  generalized feedback shift register, 525
  general plane clipping, 241, 243–244
  linear collision response, 637–638
  midpoint subdivision, 474
  random number generators, 516
  shadows, 367–368
  simple collision system performance, 593–596
  Thomas algorithm, 446
  vertex normal generation, 352–353
Alpha blending
  definition, 261
  pixels, 416–417
Alpha values, color representation
  definition, 260–261
  remapping into unit cube, 261–263
ALUs, *see* Arithmetic logic units (ALUs)
Ambient light, characteristics, 332–334
AMD processors
  basic vector class implementation, 43
  3D-specific FPus, 30
Angle of rotation, affine transformations, 141
Angles, general, trigonometric functions, A-1–A-4
Angular momentum
  rotational collision response, 639–640
  rotational dynamics, 626–628
Angular velocity, rotational dynamics, 622–625
Animation, definition, 431
Anisotropic filtering, and mipmaps, 412
Antialiasing
  application, 427–428
  method, 420–425
  supersampled antialiasing, 425–426
Application programming interface (API)
  antialiasing application, 427–428
  basic concept, 256–257
  depth buffering, 387
  and matrix decomposition, 168
  rasterization pipeline, 372
  texture samplers, 297
Approximations, real numbers, 2–3
Arbitrary point, affine transformation around, 156–158
Arc length
  computation, 477–479
  definition, 476
Arcsine, definition, A-11
Area sampling, definition, 422–423
Area under the curve, definition, B-11
Arithmetic logic units (ALUs), fixed-point number calculations, C-9
Arithmetic operations
  on functions, B-6
  IEEE 754 floating-point standard
    addition and subtraction, 13–14
    multiplication, 15
    rounding modes, 15–16
Artificial intelligence (AI) agent, dot product applications, 49–51
Augmented matrix
  and Gaussian elimination, 114
  linear equations, 114–115
Axis-aligned bounding boxes (AABBs)
  AABB–AABB intersection, 566
  AABB–plane intersection, 570–571
  AABB–ray intersection, 567–570

1. Page numbers with the letters A, B, C, and D before it refer to the Appendix pages. Page numbers with the letter X refer to the Exercise pages. Both the Appendices and Exercises appear on the CD.

Axis-aligned bounding boxes (*continued*)
basic definitions, 563–565
exercises, X-18–X-19
OBB–ray intersection, 580
simple collision system, 591
Axis–angle representation
concatenation, 184
definition, 181–182, 181–184
format conversion, 182–184
vector rotation, 184
Axis of rotation, affine transformations, 141

B

Back substitution, and Gaussian elimination, 116–117
Backward Euler methods, rigid body dynamics, 619
Barycentric coordinates
definition, 69
triangle, 85
Basis vectors
definition, 40
definition and representation, 62–63
and linear transformations, 104–106
Bayesian approach, basic probability, 496
Bell curve, definition, 504
Bernstein basis, Bézier curves, 453
Bézier curves
arc length computation, 479
and B-splines, 456–458
curve sampling, 475–476
definition, 452–454
exercises, X-16
piecewise, 455–456
spherical linear interpolation, 468
Biased numbers, definition, 10
Bilinear texture filtering, basic concept, 401–402
Binary scientific notation, rules, 6–8
Binomial coefficient, definition, 504
Binomial distribution, definition, 503
Binomial random variable, definition, 503
Bits, IEEE 754 floating-point standard, 10
Block matrices, overview, 92–93
Bluriness, and minification, 402
Boundary conditions, Hermite curves, 439
Bounding hierarchies
and scene graphs, D-6–D-8
simple collision system, 590–591
Box, inertial tensor example, 627
Box filtering, definition, 407
Brightness
definition, 317
RGB color model, 257
Brown, Robert G., 513
B-splines
curve sampling, 476
examples and characteristics, 456–458
spherical linear interpolation, 468
Buffer swapping, rasterization pipeline, 372
Bump mapping, lighting, 363

C

Cabinet projection, definition, 215
Calculus terminology
basic derivatives, B-6–B-7
continuity, B-3
derivatives, B-4–B-5
Gaussian quadrature, B-15–B-17
integral definition, B-11–B-12
integral evaluation, B-13–B-14
limits, B-1–B-3
space curves, B-17–B-19
Taylor's series, B-9–B-10
transcendental function derivatives, B-7–B-9
trapezoidal rule, B-14–B-15
Camera
definition, 205
virtual, 205–206
Camera control
basics, 488
Frenet frame, 488–489, 491
orientation, 488
parallel transport frame, 490
Camera obscura, definition, 213
Canonical random variable, definition, 499
Capsule-capsule intersection, testing, 574
Capsule-plane intersection, testing, 575
Capsule-ray intersection, testing, 575
Capsules
definition, 571
simple collision system, 590–591
Carry methods, linear congruential generators, 522–524
Cartesian coordinate system
affine frame, 68
general angles, A-1–A-4
polar coordinate conversions, 72–73
rotation, 143–144
spherical coordinate conversions, 73–75
2D and 3D systems, 64–66
Cartesian frame, affine space, 67
Catastrophic cancelation, IEEE 754 floating-point standard, 22–24
Cathode ray tubes (CRT)
color range limitation, 258–259
color storage formats, 265
operations on color, 259–260
Catmull–Rom splines
creation, 448–450
exercises, X-15–X-16
spherical linear interpolation, 468
Cavalier projection, definition, 215
Cayley, Arthur, 186
CDF, *see* Cumulative density function (CDF)
Center of mass, definition, 622–623
Center of projection, definition, 213
Center of rotation, affine transformations, 141
Central Limit Theorem
chi-square test, 507
probability distributions, 505
Central processing unit (CPU)
fixed-point number calculations, C-9
vertex buffers, 269
Centroid, point set, 69
Characteristic equation, and matrix eigenvalues, 129
Chi-square test
distribution, 509
equidistribution, 512
hypotheses, 507
theoretical frequencies, 508–509
Chopping, floating-point multiplication, 16
Clamped cubic spline, Hermite curves, 445–446
Clamping
color saturation, 261–262
texture, 304–306
Class implementation
basic vector, 42–44
vector length, 46
Clipping
general plane, 239–244
homogeneous, 244–245
purpose, 235
triangles in rendering pipeline, 276
Closest points
on line to point, intersection testing, 542–543
between two line segments, 551–553
Code, *see* Sample code
Coefficient of restitution, linear collision response, 636
Cofactor of entry, determinant computation, 123
Collinear points, definition, 79
Collision response
computation, 635
constraints, 641–642
contact generation, 630–634

linear collision response, 634–638
resting contact, 641
rotational collision response, 638–640
Collision space, into slabs, 594
Collision system
base primitive choice, 589
basic rules, 588–589
bounding hierarchies, 590–591
dynamic objects, 591–593
performance improvements, 593–596
related systems, 596–599
Coloring
basic methods, 287
per-object colors, 288
per-triangle colors, 290
per-vertex colors, 288–290
sharp edges and vertex colors, 290–291
textures as materials, 362
Color range, limitations, 258–259
Color representation
color range limitation, 258–259
colors as vectors, 257–258
operations on color, 259–260
remapping into unit cube, 261–263
RGB color model, 257
storage formats, 264–266
Colors, per-vertex attributes, 266
Color storage formats, examples, 264–266
Column major order, matrix implementation, 99–100
Column space, matrices and linear transformations, 107
Combined lighting equation
complexity, 347–348
equation definitions, 343–344
lit sphere example, 345
multi-light code, 346–347
shader code, 344
Compiling, fragment shaders, 284–285
Complementary angle, definition, A-7–A-8
Complete fragments, definition, 373
Components, vectors, 41
Composite functions, definition, B-6–B-7
Composition, linear transformations, 108
Compound angle, definition, A-9
Computer code, *see* Sample code
Computer number representation
fixed point
addition and subtraction, C-9
basic concept, C-5
basic representation, C-5–C-6
division, C-11–C-12
limits, C-16
multiplication, C-10–C-11
overflow and underflow, C-13–C-16
range and precision, C-7–C-9
real-world fixed point, C-12–C-13
floating point numbers, 4–6
IEEE 754 floating-point standard, 9–25
integral representation finiteness, C-1–C-2
overflow, C-3–C-4
overview, 1–2
range overview, C-2–C-3
range and type conversion, C-4–C-5
real numbers, 2–4
real-world floating-point, 25–32
Concatenation
affine transformations, 167
axis–angle representation, 184
fixed and Euler angles, 178
linear transformations, 108
quaternions, 193–195
world matrix code, 253
Concave polygon, definition, 83
Conformable for addition, definition, 90
Conjugate, quaternions, 196
Constraint force, collision response, 641
Constraints, collision response, 641–642
Contact generation, collision response, 630–634
Continuity, calculus overview, B-3
Continuous, definition, 433, B-3
Continuous random variable, definition, 499
Control points, Bézier curves, 452
Convex combination, point set, 69
Convex hull, point set, 69
Convex polygon, definition, 83
Convex set, points, 69
Coordinate frame, affine space, 67
Coplanar points, definition, 82
Cosecant
general angles, A-3
right triangle, A-1
Cosine
as even function, A-8
right triangle, A-1
Cotangent
general angles, A-3
right triangle, A-1
Covariance matrix, object-oriented bounding boxes, 577
CPU, *see* Central processing unit (CPU)
Cramer's method
for inverse computation, 129
triangle–ray intersection, 586–587
Cross product
definition, 53
implementation, 54
for parallel vectors, 55
CRT, *see* Cathode ray tubes (CRT)
Cubic methods, spherical linear interpolation, 468
Culling
frustrum, and collision systems, 597–599
method, 238–239
purpose, 235
Cumulative density function (CDF)
definition, 501
nonuniform random number distributions, 529–530
Curvature
definition, 434
space curves, B-18–B-19
Curves, speed control
constant speed, 480–485
variable speed, 485–488
Curve sampling
basic concept, 470–471
forward differencing, 471–473
midpoint subdivision, 473–476
Cylinder, inertial tensor example, 627

D

De Casteljau's method, curve sampling, 475
De Fermat, Pierre, 494
Definite integral, definition, B-11
Deformations
affine transformations, 137
scaling, 150–151
Degenerate triangle, definition, 83
Degree of freedom, fixed and Euler angles, 179–181
De Mere, Chevalier, 495, 496, 498
Denormalized numbers, *see* Denormals
Denormals
floating-point standards, 21
floating-point zero values, 16
real-world performance, 27–28
Dependent events, basic probability, 496
Dependent trials, basic probability, 496
Depth buffering
application, 387–388
basic concept, 378–381
numerical precision, 385–387
per-fragment depth value computation, 381–385
and pixel blending, 417–419
Depth complexity, fragments, 373–375
Depth cuing, exercises, X-11–X-12
Derivatives
associated definitions, B-4–B-5
basic derivatives, B-6–B-7
transcendental functions, B-7–B-9

Descartes, Rene, Cartesian coordinates, 64
Determinants
  adjoint matrix and inverse, 128–129
  computation, 123–126
  definition, 121–123
  and elementary row operations, 126–128
df/dx, definition, B-4
Diagonalization, matrices, 130
Diagonal matrix, definition, 89
DIEHARD, chi-square test, 513
DieHarder, chi-square test, 513
Diffuse color, textures as materials, 362
Diffuse lighting, characteristics, 334–337
Diffuse map, definition, 362
Dimension, basis vectors, 63
Direct3D
  antialiasing application, 428
  basic concept, 256–257
  fragment shader application, 285
  HLSL exercises, X-14
  light approximation, basics, 316
  light interaction, 331
  matrix exercises, X-8
  object vertices, 268
  oblique parallel projection, 233–234
  oblique perspective matrix, 230
  orthographic parallel projection, 232
  perspective projection, 227
  pixel blending, 419
  rendering pipelines, 277
  simple lighting approximation, 318–319
  texture images, 294
  viewing transformation management, 253
Directional lights, characteristics, 320–321
DirectX
  vector representation, 92
  vertex shader inputs, 281
Discontinuous, definition, B-3
Discrete random variables, definition, 498
Disc sampling, random numbers, 533–535
Displays
  as hardware, 370–371
  raster displays, 369
Distance attenuation, *see* Inverse-square law
Distance testing
  line-line distance, 550–551
  line-point distance, 544–545
  line segment-line segment distance, 553
  line segment-point distance, 546–547
Distance-time function, variable curve speed, 485–488
Distribution
  binomial distribution, 503
  chi-square test, 509
  Gaussian distribution, 504
  nonuniform, random numbers, 529–530
  probability distribution function, 499
  standard normal distribution, 504
Division, fixed-point numbers, C-11–C-12
Domain, linear transformation definition, 101
Dot product
  algebraic rules, 48
  definition, 47
  equations, 47–48
  library implementation, 51
  perpendicular and orthogonal vectors, 48–49
  quaternions, 189
  testing vector angles, 49–51
Double angle, definition, A-9
Double precision, IEEE 754 floating-point standard, 9, 24–25
Dynamic objects, simple collision system, 591–593
Dynamics
  definition, 602
  linear dynamics, 602–609
  rigid body, *see* Rigid body dynamics
  rotational dynamics, 622–630

E

Ease-in/ease-out, variable curve speed, 485–488
Efficiency, rigid body dynamics, 643–644
Eigenspace, matrices, 130
Eigenvalues, as matrix property, 129–130
Eigenvectors, as matrix property, 129–130
8-bit value, binary scientific notation, 7–8
Elementary matrices
  and determinants, 126–128
  linear system solutions, 113
Elementary outcome, basic probability, 495
Elementary row operations, linear system solutions, 113
Emissive color, textures as materials, 362
Emissive light, characteristics, 332
Emulation, *see* Software emulation
Environment mapping, applications, 366–367
Equal, matrices, 89
Equidistribution test, chi-square, 512
Error, real number representation, 3–4
Euclid
  line definition, 75
  plane definition, 80
  points, 64
Euclidean distance, affine space, 68
Euclidean inner product, *see* Dot product
Euler angles
  concatenation, 178
  definition, 174–176
  format conversion, 177–178
  gimbal lock issues, 179–181
  vector rotation, 178–179
Euler methods
  backward, 619
  exercises, X-19
  modified method, 615
  rigid body dynamics, 611–614
  rotational quantity integration, 628–630
  semi-implicit, 621–622
Even functions, trigonometry overview, A-8
Event, basic probability, 495
Exercises
  affine transformations, X-6
  floating-point numbers, X-1–X-2
  geometry, X-9–X-11
  interpolation, X-15–X-16
  intersection testing, X-18–X-19
  lighting, X-11–X-13
  linear transformations, X-5
  matrix operations, X-4–X-6
  orientation representation, X-7–X-8
  programmable shading, X-9–X-11
  projection, X-8–X-9
  random numbers, X-16–X-17
  rasterization, X-13–X-14
  rigid body dynamics, X-19–X-20
  vectors, X-3–X-4
  viewing, X-8–X-9
Expected value, definition, 502
Explicit methods, rigid body dynamics, 619
Exponent
  binary scientific notation, 6–8
  catastrophic cancelation, 23–24
  double precision, 24
  floating-point infinity values, 18
  floating-point nonnumeric values, 19
  floating-point zero values, 16–17
  IEEE 754 floating-point standard, 9–11
  restricted scientific notation, 5–6
  scientific notation, 4

Exponentials, calculus overview, B-8–B-9
Eye, and object projection, 236
Eyepoint, definition, 205

F

Faceted shading, definition, 290
Favorable outcome, basic probability, 495
Fibonacci series
  generalized feedback shift register algorithms, 525
  lagged methods, 521–522
  random number generators, 516–517
Field of view, view frustrum, 217–218
Fill convention, partial fragment handling, 377
Filtering, color operations, 260
Finiteness, integral representation, C-1–C-2
First derivative, definition, B-4
Fishtank VR, oblique perspective, 231
Fixed angles
  concatenation, 178
  definition, 174–176
  format conversion, 177–178
  gimbal lock issues, 179–181
  vector rotation, 178–179
Fixed-function rendering pipeline *vs.* programmable, 277–278
Fixed-point numbers
  addition and subtraction, C-9
  basic concept, C-5
  basic representation, C-5–C-6
  binary scientific notation, 7–8
  division, C-11–C-12
  limits, C-16
  multiplication, C-10–C-11
  overflow and underflow, C-13–C-16
  range and precision, C-7–C-9
  from real numbers, C-8–C-9
  real-world fixed point, C-12–C-13
Flat shading, definition, 290
Floating-point addition
  basic concepts, 13–14
  catastrophic cancelation, 22
Floating-point numbers
  exercises, X-1–X-2
  generation, 529
  IEEE 754 standard
    addition and subtraction, 13–14
    basic representation, 9–11
    catastrophic cancelation, 22–24
    denormals and gradual underflow, 21
    double precision, 24–25
    infinity, 17–18
    nonnumeric values, 18–19
    overview, 9
    range and precision, 11–13
    very small values, 19–21
  real-world, *see* Real-world floating-point
  scientific notation, 4–5
Floating-point subtraction
  basic concepts, 13–14
  catastrophic cancelation, 22
  example, 20–21
Floating-point units (FPUs)
  denormals and gradual underflow, 21
  internal FPU precision, 25–26
  software emulation, 28–29
  3D-specific, 30–31
Flush-to-zero
  definition, 20
  implementation, 21
Fogging, exercises, X-11–X-12
Forces, linear dynamics, 605–606
Format conversion
  axis–angle representation, 182–184
  fixed and Euler angles, 177–178
  quaternions, 189–193
Forward differencing
  curve sampling, 471–473
  definition, 384
FPUs, *see* Floating-point units (FPUs)
Fractional texel coordinate, computation, 398, 400
Fragments
  definition, 373
  to pixels
    antialiasing application, 427–428
    antialiasing method, 420–425
    basic approach, 415–416
    multisampled antialiasing, 426–427
    pixel blending application, 419
    pixel blending and depth buffering, 417–419
    pixel blending methods, 416–417
    supersampled antialiasing, 425–426
  rendering pipeline, 276
  texturing with mipmap, 408–411
  from triangles, 375–376
Fragment shaders (FS)
  compiling, linking, applications, 284–285
  evaluation, 395
  function, 278–279
  input computation, 388–389
  inputs, 283
  link to vertex shaders, 282–283
  outputs, 284
  per-vertex attributes, 389–392
  random number noise, 537
  texture coordinate interpolation, 392–394
  texture coordinate sources, 394–395
  uniform values, 389
  uniform value setting, 286–287
Framebuffers, as hardware, 370–371
Frenet frame, camera control, 488–489, 491
Frustrum culling, and collision systems, 597–599
FS, *see* Fragment shaders (FS)
Function
  arithmetic operations on, B-6
  trigonometric function derivatives, B-7–B-8
Fundamental theorem of calculus, basic statement, B-12

G

Game Developer Magazine, 370, 378
Game objects, manipulation via affine transformations
  model and world coordinate frames, 160–161
  overview, 159–160
  placement, 161–164
Gaussian distribution, definition, 504
Gaussian elimination
  and determinants, 126–127
  Hermite curves, 446
  linear equation systems, 113–117
  matrix inverse, 119
Gaussian quadrature
  arc length computation, 477–478
  definition, B-15–B-17
Gauss-Jordan elimination
  definition, 116
  matrix inverse, 119–120
General angles, trigonometric functions, A-1–A-4
Generalized feedback shift register (GFSR) algorithm, Mersenne Twister, 525
Generalized line equation, characteristics, 77–79
Generalized plane equation, representation, 80–82
General plane clipping, methods, 239–244
General polynomials, definition, B-7
Geometric objects, vertices, 267–270
Geometric vectors, definition, 36–39
Geometry
  drawing, 274–275
  exercises, X-9–X-11
  per-fragment lighting, 355–356
  points as, 64–66
  spotlights, 327
  visible geometry, 378, 381–388
GFSR algorithm, *see* Generalized feedback shift register (GFSR) algorithm
Gimbal lock, fixed and Euler angles, 179–181

Global illumination, definition, 318
Gloss map, definition, 362
Glow map, definition, 362
GL shading language (GLSL)
  definition, 278
  fragment shader application, 284–285
  fragment shader inputs, 283
  fragment shader outputs, 284
  input and output values, 279
  inputs, 280–281
  OpenGL exercises, X-14
  random number noise, 538–539
  simple lighting approximation, 319
Gouraud shading
  definition, 288–289
  limitations, 292
  per-fragment lighting, 354–355
  per-vertex attributes, 391–392
  sharp edges, 353–354
  and texturing, 309–310
  as vertex-centric method, 350–351
GPUs, *see* Graphics processing units (GPUs)
Gradient noise, random numbers, 537
Gradual underflow, floating-point standards, 21
Gram–Schmidt orthogonalization
  equations, 51–52
  interpolation of orientation, 461
Graphics pipeline, definition, 203–204
Graphics processing units (GPUs)
  index buffers, 274
  real-world floating-point, 31–32
  texture images, 295
  textures as materials, 362
  vertex buffers, 269
  vertices into triangles, 273
Guard band, definition, 238

H

Half-angle, definition, A-10–A-11
Half-precision floating-point formats, characteristics, 31–32
Hamilton, Sir William, 185
HDR rendering, *see* High dynamic range (HDR) rendering
Heading, as Euler angle, 175–176
Heavy point, vertex definition, 266
Hermite curves
  automatic generation, 444–446
  and Bézier curves, 452–454
  Catmull-Rom splines, 448–449
  definition, 438–441
  exercises, X-15–X-16
  natural end conditions, 447–448
  and piecewise Bézier curves, 455
  tangent manipulation, 441–444
Heun's method, rigid body dynamics, 616
High color, color storage formats, 265
High dynamic range (HDR) rendering, color remapping, 263
High-level shading language (HLSL)
  definition, 278
  Direct3D exercises, X-14
  random number noise, 538–539
  simple lighting approximation, 319
Hole at Zero
  definition, 20
  IEEE 754 floating-point standard, 19–21
Homogeneous clipping
  methods, 244–245
  screen transformation, 246
Hypothesis testing, chi-square test, 507

I

Identity, quaternions, 195–197
Identity matrix, overview, 96–97
IEEE, Institute of Electrical and Electronic Engineers (IEEE)
Illuminance
  definition, 317
  diffuse lighting, 334–336
Image-based texture mapping, overview, 292–293
Image lookup, for shading, 293–294
Implementation
  basic vector, 42–44
  column major order, 99–100
  cross product, 54
  definition, 256
  dot product, 51
  flush-to-zero, 21
  matrix, 98–101
  point via OpenGL, 70–71
  points, 70–72
  row major order, 98–99
  scalar multiplication, 43–44
  scene graphs, D-1–D-5
  SSE, 43
  vector addition, 43
  vector length, 46
  vectors, 42–44
Implicit methods, rigid body dynamics, 619–621
Impulse-based system, linear collision response, 634
Increment, definition, 518
Indefinite integral, definition, B-11
Independent events, basic probability, 496
Independent trials, basic probability, 496
Index array, definition, 272
Index buffers, definition, 274
Indirect texturing, definition, 395
Inelasticity, computation, 636
Inertia tensor, rotational dynamics, 626–628
Infinite viewer approximation, specular highlight, 342–343
Infinity, IEEE 754 floating-point standard, 17–18
Initial value problems, definition, 609
Instance, game objects, 160
Institute of Electrical and Electronic Engineers (IEEE) standard
  IEEE 754 floating-point standard
    addition and subtraction, 13–14
    basic representation, 9–11
    catastrophic cancelation, 22–24
    denormals and gradual underflow, 21
    double precision, 24–25
    infinity, 17–18
    multiplication, 15
    nonnumeric values, 18–19
    overview, 9
    range and precision, 11–13
    rounding modes, 15–16
    very small values, 19–21
    zero values, 16–17
  Intel's SSE, 29–30
Integers, random numbers, 528–529
Integer texel coordinate, definition, 396
Integral representation, finiteness, C-1–C-2
Integrals, evaluation, B-13–B-14
Intel, SSE, 29–30
Interpolation
  arc length computation, 476–479
  camera control, 488–491
  curve sampling
    basic concept, 469–470
    forward differencing, 471–473
    midpoint subdivision, 473–476
  exercises, X-15–X-16
  linear
    definition, 435–436
    piecewise, 436–438
  overview, 431–433
  speed along curve
    constant speed, 480–485
    variable speed, 485–488
Interpolation of orientation
  basic concept, 458–461
  linear interpolation, 461–465
  performance improvements, 469–470
  spherical linear interpolation, 465–468
Interpolation of position
  Bézier curves, definition, 452–454
  B-splines, 456–458
  Catmull-Rom splines, 448–450
  definitions, 433–434

Hermite curves
automatic generation, 444–446
definition, 438–441
natural end conditions, 447–448
tangent manipulation, 441–444
Kochanek–Bartels splines, 450–452
piecewise Bézier curves, 455–456
Intersection testing
AABB definitions, 563–565
AABB-AABB intersection, 566
AABB-plane intersection, 570–571
AABB-ray intersection, 567–570
capsule-capsule intersection, 574
capsule-plane intersection, 575
capsule-ray intersection, 575
closest point on line to point, 542–543
closest point on line segment to point, 544–545
closest points between two lines, 548–550
closest points between two line segments, 551–553
exercises, X-18–X-19
general linear components, 554
line-line distance, 550–551
line-point distance, 544–545
line segment-line segment distance, 553
line segment-point distance, 546–547
OBB definitions, 576–578
OBB-OBB intersection, 578–580
OBB-plane intersection, 582–583
OBB-ray intersection, 580–582
object intersection overview, 554–556
overview, 541–542
simple collision system
base primitive choice, 589
basic rules, 588–589
bounding hierarchies, 590–591
dynamic objects, 591–593
performance improvements, 593–596
related systems, 596–599
sphere definitions, 556–560
sphere-plane intersection, 563
sphere-sphere intersection, 560–561
swept spheres definitions, 571–574
triangle-plane intersection, 588
triangle-ray intersection, 585–588
triangle-triangle intersection, 583–585
Inverses
calculus overview, B-8
definition, A-11
and determinants, 128–129
quaternions, 195–197
Inverse shear transformation, equation, 156
Inverse-square law, point lights, 323–325
Irradiance, definition, 318
Isotropic filtering, and mipmaps, 412

J

Jacobi method, matrix computations, 130
JND, *see* Just noticeable difference (JND)
Just noticeable difference (JND), color storage formats, 266

K

Kernel, *see* Null space
Kinematics, definition, 603
Knots, and B-splines, 457
Kochanek-Bartels splines, creation, 450–452

L

Lagged Fibonacci methods, linear congruential generators, 521–522
Lambertian reflector, definition, 334
Law of cosines, 47, A-6
Law of large numbers, definition, 495
Law of sines, triangles, A-5–A-6
Law of tangents, triangles, A-6–A-7
LCDs, *see* Liquid crystal displays (LCDs)
LCG, *see* Linear congruential generator (LCG)
Leapfrog Verlet method, rigid body dynamics, 617–618
Legendre polynomials, definition, B-16
Leibnitz notation, definition, B-4
Length, vectors, 44–47
Library implementation, dot product, 51
Light
measurement, 317–318
as ray, 318
Light approximation
basics, 316
simple, 318–319
Light categories
ambient, 332–334
diffuse, 334–337
emission, 332
specular, 338–343
Light direction vector, definition, 319
Lighting
combined lighting equation, 343–348
exercises, X-11–X-13
flat-shaded lighting, 349–350
normal mapping, 363–366
overview, 315–316
per-fragment lighting, 354–358
per-vertex lighting, 350–354
reflective objects, 366–367
and shading, 348
shadows, 367–368
and textures
basic modulation, 359–360
relationship, 358
specular lighting, 360–361
textures as materials, 362–363
Light intensity value, definition, 319
Light interaction, and surface materials, 331–332
Light sources
directional lights, 320–321
point lights, 321–326
shadows, 367
spotlights, 327–330
types, 319–320, 330–331
Limits
calculus overview, B-1–B-3
fixed-point numbers, C-16
Linear-bilinear filtering, and mipmaps, 411–412
Linear collision response, methods, 634–638
Linear combinations, definition, 39–40
Linear components, intersection testing, 554
Linear congruential generator (LCG)
definition, 517–518
modulus choice, 518–520
multiplier choice, 520–521
Linear dependence, definition, 39
Linear dynamics
constant acceleration, 602–605
forces, 605–606
momentum, 606
variable acceleration, 607–609
Linear equation systems
associated definitions, 110–112
Gaussian elimination, 113–117
Linear independence, definition, 39
Linear interpolation
definition, 435–436
as orientation, 461–465
piecewise, 436–438
spherical, 465–468
Linear momentum, rigid body dynamics, 606
Linear space, definition, 59
Linear transformations
affine transformations, 136–137, 138
associated definitions, 101–103
and basis vectors, 104–106
combining, 108–110

Linear transformations (*continued*)
definition, 87
exercises, X-5
and matrices, 106–108
null space and range, 103–104
Line-line distance, testing, 550–551
Line-point distance, intersection testing, 544–545
Line of projection, definition, 213
Lines
closest point to point, intersection testing, 542–543
closest points between, 548–550
definition, 75
generalized equation, 77–79
parameterized, 76–77
Line segment-line segment distance, intersection testing, 553
Line segment-point distance, intersection testing, 546–547
Line segments
closest point to point, 544–545
closest points between, 551–553
definition, 76
Line-sphere intersection, example, 562
Linking, fragment shaders, 284–285
Liquid crystal displays (LCDs)
color range limitation, 258
and color storage formats, 265
Local space, *see* Model frames
LOD selection methods, exercises, X-14
Logarithms, calculus overview, B-8–B-9
Lower triangular matrix, definition, 89
Lumen, definition, 317
Luminance
as color operation, 259–260
definition, 317, 318
Luminous flux, definition, 317
Luminous flux density, definition, 317
Lux, definition, 317

M

Magnitude, quaternions, 188–189
Main diagonal, definition, 89
Manhattan distance, as norm example, 44–47
Mantissa
binary scientific notation, 6–8
catastrophic cancelation, 23–24
double precision, 24
floating-point infinity values, 18
floating-point multiplication, 15
floating-point nonnumeric values, 19
floating-point zero values, 16–17
IEEE 754 floating-point standard, 9–11
normalized, 19–21
restricted scientific notation, 5–6
scientific notation, 4
Mapping
bump mapping, 363
diffuse map, 362
environment mapping, 366–367
gloss map, 362
glow map, 362
image-based texture mapping, 292–293
mipmapping, 404–415
nearest-neighbor texture mapping, 310, 397
normal mapping, 363–366
texture mapping, 292–297, 308, 310
tone mapping, 262–263
Master, game objects, 160
Materials, textures as, 362–363
Matrix
adjoint matrix, 128–129
affine transformation definition, 134–136
augmented matrix, 114–115
block matrices, 92–93
covariance matrix, 577
definition, 87
diagonal matrix, 89
eigenvalues and eigenvectors, 129–130
exercises, X-4–X-6, X-8
Gaussian, 113–117
Hermite curves, 446
identity matrix, 96–97
implementation, 98–101
linear equation definitions, 110–112
linear equation solving, 112–113
linear interpolation, 435–436
and linear transformations, 106–108
lower triangular matrix, 89
orthogonal matrix, 120, 145–146
overview, 88–89
into quaternion, 191–192
simple operations, 90–91
skew symmetric matrix, 91
square matrix, 88
symmetric matrix, 91
triangular matrix, 89
tridiagonal matrix, 446
twist matrix, 525–526
upper triangular matrix, 89
vector operations, 97–98
vector representation, 92
zero matrix, 88
Matrix addition
as operation, 90
test code, 101
Matrix decomposition
affine transformations, 164–166
avoidance, 166–169
Matrix inverse
definition, 117–120
simple inverses, 120–121
Matrix-matrix multiplication, test code, 100
Matrix product, overview, 94–96
Matrix-vector multiplication
with term expansion, 106
test code, 100
Matrix-vector notation
example, 95
multiplication, 100
Mean, calculation, 502–503
Memory, and double precision, 25
Mersenne Twister
concept and developments, 524–525
twisted generalized feedback shift register algorithm, 526–527
twist matrix, 525–526
Micro-impulse engine, collision response, 641
Midpoint method
definition, 614
exercises, X-19
Midpoint subdivision, curve sampling, 473–476
Minification, textures, 402–404
Minor of entry, determinant computation, 123
Mipmapping
application, 412–415
basic concept, 404–407
definition, 404
fragment texturing, 408–411
and texture filtering, 411–412
Mixed congruential method, definition, 518
Model, definition, 35
Model frames, game objects, 160–161
Model-to-world transformation, definition, 161
Modified Euler's method, rigid body dynamics, 615
Modulate mode, pixel blending, 418
Modulate with late addition, definition, 360
Modulus
choice in LCG, 518–520
definition, 518
Moments of inertia, rotational dynamics, 627
Momentum, rigid body dynamics, 606
Monte Carlo integration, RNGs, 514
Mother of All Random Number Generators, 523
Movement
with constant acceleration, 602–605
with variable acceleration, 607–609
MRTs, *see* Multiple render targets (MRTs)

MSAA, *see* Multisampled antialiasing (MSAA)
Multiple render targets (MRTs), fragment shader outputs, 284
Multiplication
  fixed-point numbers, C-10–C-11, C-13–C-14
  IEEE 754 floating-point standard, 15
  matrix–matrix multiplication, 100
  matrix–vector multiplication, 106
  matrix–vector notation, 100
  quaternions, 194–195
  scalar, *see* Scalar multiplication
Multiplicative congruential method, definition, 518
Multipliers
  choice in LCG, 520–521
  definition, 518
Multisampled antialiasing (MSAA)
  antialiasing application, 427–428
  method, 426–427

N

Natural end conditions, Hermite curves, 447–448
Natural spline, Hermite curves, 447
NDC, *see* Normalized device coordinates (NDC)
Nearest-neighbor texture mapping
  definition, 310
  magnification, 397
Negation, quaternions, 188
Newton-Raphson root finding, constant speed, 480–485
Noise, random numbers, 535–539
Nonnumeric values, IEEE 754 floating-point standard, 18–19
Nonuniform B-splines (NURBS), definition, 457–458
Nonuniform distributions, random numbers, 529–530
Normal force, linear dynamics, 605
Normalization, quaternions, 188–189
Normalized device coordinates (NDC)
  characteristics, 216
  fragment shaders, 389, 391
  matrix exercises, X-8
  oblique perspective, 228
  perspective projection, 223–224
  picking, 250
  screen transformation, overview, 245–248
  viewing transformation management, 252–253
Normalized vector
  definition, 36
  and vector length, 45–46
Normal mapping
  lighting, 363–365
  map generation, 365–366
Normal-point form, generalized plane equation, 80–82
Normal vector, vertex, 267
Norms, and vector length, 44
NTSC, pixel aspect ratio, 249
Null hypothesis, chi-square test, 507
Nullity, and linear transformations, 103
Null space, and linear transformations, 103–104
Numerical integration, rigid body dynamics
  basic definitions, 609–610
  Euler's method, 611–614
  implicit methods, 619–621
  Runge-Kutta methods, 614–616
  semi-implicit methods, 621–622
  Verlet methods, 616–618
NURBS, *see* Nonuniform B-splines (NURBS)

O

OBBs, *see* Object-oriented bounding boxes (OBBs)
Object hierarchies, examples, 169–171
Object intersection
  AABB-AABB intersection, 566
  AABB-plane intersection, 570–571
  AABB-ray intersection, 567–570
  axis-aligned bounding boxes, basic definitions, 563–565
  capsule-capsule intersection, 574
  capsule-plane intersection, 575
  capsule-ray intersection, 575
  OBB definitions, 576–578
  OBB-OBB intersection, 578–580
  OBB-plane intersection, 582–583
  OBB-ray intersection, 580–582
  overview, 554–556
  sphere definitions, 556–560
  sphere-plane intersection, 563
  sphere-ray intersection, 561–562
  sphere-sphere intersection, 560–561
  swept spheres definitions, 571–574
  triangle-plane intersection, 588
  triangle-ray intersection, 585–588
  triangle-triangle intersection, 583–585
Object-oriented bounding boxes (OBBs)
  basic definitions, 576–578
  exercises, X-18–X-19
  OBB-OBB intersection, 578–580
  OBB-plane intersection, 582–583
  OBB-ray intersection, 580–582
  simple collision system, 589, 591
Objects
  pixel blending, 419
  texture coordinate mapping, 298–300
Object space, *see* Model frames
Oblique parallel projection, construction, 232–234
Oblique perspective, construction, 228–231
Oblique projection, definition, 215
Odd functions, trigonometry overview, A-8
Once per light, computation, 322
OpenGL
  antialiasing application, 428
  basic concept, 256–257
  camera control, 210
  fragment shader application, 284–285
  and GPUs, 31
  GLSL exercises, X-14
  light approximation, basics, 316
  light interaction, 331
  matrix exercises, X-8
  oblique parallel projection, 233
  oblique perspective, 228
  orthographic parallel projection, 231
  perspective projection, 227
  picking, 249
  pixel blending, 419
  point implementation, 70–71
  random number noise, 537
  rendering pipelines, 277–278
  simple lighting approximation, 318–319
  varying values, 389
  vector representation, 92
  vertex shader inputs, 281
  viewing transformation management, 252–254
  view-to-world transformation, 206–208
  and virtual camera definition, 206
  world-to-view transformation, 211
Operator, linear transformation definition, 102
Orbit
  Euler method, 611, 613
  numerical integration example, 610
Orientation
  camera control, 488
  vs. rotation, 149–150
  rotational dynamics, 622–625
Orientation representation
  axis–angle representation, 181–185
  exercises, X-7–X-8
  fixed and Euler angles, 174–181
  overview, 173–174
  quaternions, 185–201
  rotation matrices, 174
Origin
  affine space, 67
  definition, 64
Orthogonal matrix
  example, 120
  and rotations, 145–146

Orthogonal vectors, and dot product, 48–49
Orthographic parallel projection, construction, 231–232
Orthographic projection, definition, 215
Orthonormal, definition, 51–52
Output processing, rendering pipeline, 276
Overdraw, fragments, 375
Overflow
  fixed-point numbers, C-13–C-16
  number representation, C-3–C-4
Overshooting, Kochanek-Bartels splines, 451

P

Parallel, vectors, 39
Parallelogram, determinant, 122–123
Parallelopiped, determinant, 122–123
Parallel projection
  characteristics, 214–215
  orthographic, 231–232
  view frustrum, 219
Parallel transport frame, camera control, 490
Parameter, and lines, 76
Parameterized lines, characteristics, 76–77
Parameterized planes, characteristics, 80
Parametric curves, definition, 433
Partial fragments
  definition, 373
  handling, 376–378
Partial pivot, and Gaussian elimination, 114
Pascal, Blaise, 494
PDF, *see* Probability density function (PDF)
Pearson's chi-square test, definition, 508
Penetration, collision response, 632–633
Penumbra
  definition, 331
  shadows, 368
Performance
  and double precision, 25
  real-world floating-point denormals, 27–28
  overview, 26–27
  simple collision system, 593–596
Per-fragment depth values, computation, 381–385
Per-fragment lighting
  geometry, 355–356
  overview, 354–355
  Phong shading, 356–357
  shader code example, 357–358
Period, random number generators, 517
Per-object colors, coloring, 288
Perpendicular dot product, definition, 55
Perpendicular vectors
  and cross product, 55
  and dot product, 48–49
Perspective projection
  construction, 221–228
  definition, 213
Per-triangle colors, coloring, 290
Per-triangle mipmapping, definition, 410
Per-vertex colors, coloring, 288–290
Per-vertex lighting
  basics, 350–351
  sharp edges, 353–354
  vertex normal generation, 351–353
Per-vertex operations
  attributes, 266–267
  rendering pipeline, 275–276
Per-vertex values, fragment shaders, 389–392
Phong shading, definition, 356–357
Picking
  definition, 204
  methods, 249–251
Pick ray, example, 250
Piecewise Bézier curves, example, 455–456
Piecewise curve, definition, 434
Piecewise Hermite curves, examples, 441–443
Piecewise interpolation, characteristics, 436–438
Piecewise smooth, space curves, B-18
Pitch, as Euler angle, 175–176
Pivot, and Gaussian elimination, 114
Pixel aspect ratio, screen transformation, 248–249
Pixel blending
  application, 419
  and depth buffering, 417–419
  methods, 416–417
Pixels
  definition, 369, 371
  from fragments
    antialiasing application, 427–428
    antialiasing method, 420–425
    basic approach, 415–416
    multisampled antialiasing, 426–427
    pixel blending application, 419
    pixel blending and depth buffering, 417–419
    pixel blending methods, 416–417
    supersampled antialiasing, 425–426
Pixel shaders, function, 278–279
Planes
  AABB-plane intersection, 570–571
  general clipping, 239–244
  generalized equation, 80–82
  OBB-plane intersection, 582–583
  parameterized, 80
  plane normal transformation, 158–159
  sphere-plane intersection, 563
  triangle-plane intersection, 588
  triangle vertices, 84
Platform issues, and double precision, 25
Point clouds, and surface ambiguity, 270–271
Point of collision, response, 631
Point at infinity, directional lights, 320
Point lights, as light sources, 321–326
Points
  affine combination, 68–69
  affinely independent points, 69
  affine spaces, 66–68
  arbitrary point, 156–158
  basic concept, 35
  closest between two lines, 548–550
  closest between two line segments, 551–553
  closest point on line, intersection testing, 542–543
  closest point on line segment, intersection testing, 544–545
  collinear, 79
  control points, 452
  coplanar, 82
  as geometry, 64–66
  implementation, 70–72
  multiple, collision response, 643
  polar coordinates, 72–73
  radix points, 13
  simplex points, 69
  spherical coordinates, 73–75
  vertex as heavy point, 266
Point sampling, definition, 376–377
Poker hand test, chi-square, 512
Polar coordinates
  Cartesian coordinate conversion, 72–73
  rotation, 143–144
Polar decomposition, affine transformations, 165–166
Polygons, characteristics, 83–85
Polynomials
  definition, B-7
  Legendre polynomials, B-16
Position, linear dynamics, 604
Power of a variable, definition, B-6
Precision
  fixed-point numbers, C-7–C-9, C-14–C-16
  floating-point numbers, 12
  IEEE 754 floating-point standard, 11–13
  internal FPU precision, 25–26

real number representation, 3–4
z-buffering, 385–387
Predictor-corrector method, rigid body dynamics, 620
Primary colors, RGB model, 257
Primitive element
LCG multiplier, 520
simple collision system, 589
Primitive processing, rendering pipeline, 275
Principle axes, rotational dynamics, 627
Probability
basic probability, 495–498
definition, 495
history, 494–495
mean, 502–503
random variables, 498–501
special distributions, 503–506
standard deviation, 502–503
Probability density function (PDF)
definition, 500–501
nonuniform random number distributions, 530
Probability distribution function, definition, 499
Probability mass function, definition, 498–499
Products of inertia, rotational dynamics, 627
Programmable rendering pipeline
exercises, X-9–X-11
vs. fixed-function, 277–278
Projected Gauss-Seidel method, collision response, 643
Projection
definition, 212
dot product example, 50–51
exercises, X-8–X-9
overview, 203–204
Projection plane, definition, 213
Projective transformation
definition, 212–216
homogeneous coordinates, 220–221
normalized device coordinates, 216
oblique parallel projection, 232–234
oblique perspective, 228–231
orthographic parallel projection, 231–232
perspective projection, 221–228
view frustrum, 216–220
Pure rotation, affine transformations, 142
p-value, chi-square distribution, 509, 511–512
Pythagorean identities
definition, A-7
double angle, A-9
Pythagorean theorem, 45
basic equation, A-5–A-6

Q

Quadrilaterals (Quads), definition, 83
Quaternions
addition, 187
concatenation, 193–195
definition, 185–186
dot product, 189
exercises, X-7–X-8
format conversion, 189–193
identity, 195–197
interpolation of orientation, 464
inverse, 195–197
magnitude, 188–189
multiplication, 194–195
negation, 188
normalization, 188–189
rotational quantity integration, 629–630
as rotations, 186–187
scalar multiplication, 187
shortest path of rotation, 199–200
spherical linear interpolation, 466–467
and transformations, 200–201
vector rotation, 197–199

R

Radiance, definition, 318
Radius, pixel, 409
Radix points, floating-point arithmetic, 13
Random experiment, basic probability, 495
Randomness
chi-square test, 507–513
definition, 506–507
spectral test, 513–514
Random number generators (RNGs)
algorithms, 516
definition, 514
Fibonacci series, 516–517
generated points examples, 515
integers and ranges, 528
lagged Fibonacci methods, 521–522
linear congruential methods
definition, 517–518
modulus choice, 518–520
multiplier choice, 520–521
Mersenne Twister, 524–527
period, 517
spectral test, 513–514
Random numbers
basic probability, 495–498
disc sampling, 533–535
exercises, X-16–X-17
floating-point numbers, 529
integers and integer ranges, 528–529
mean, 502–503
noise, 535–539
nonuniform distributions, 529–530
overview, 494
special probability distributions, 503–506
spherical sampling, 531–533
standard deviation, 502–503
turbulence, 537–539
Random trial, basic probability, 495
Random variables, probability, 498–501
Range
fixed-point numbers, C-7–C-9
IEEE 754 floating-point standard, 11–13
linear transformation
definition, 101
and linear transformations, 103–104
number representation
overflow, C-3–C-4
overview, C-2–C-3
and type conversion, C-4–C-5
Rank
and linear transformations, 103
matrices and linear transformations, 107
Raster displays, definition, 369
Rasterization
conceptual pipeline, 371–372
coordinate mapping to texel
basic concept, 396
texture magnification, 396–402
texture minification, 402–404
definition, 235, 369
depth complexity, 373–375
display hardware, 370–371
exercises, X-13–X-14
fragments, 373
fragment shader evaluation, 395
fragment shader inputs
computation, 388–389
per-vertex attributes, 389–392
texture coordinate interpolation, 392–394
texture coordinate sources, 394–395
uniform values, 389
fragments to pixels
antialiasing application, 427–428
antialiasing method, 420–425
basic approach, 415–416
multisampled antialiasing, 426–427
pixel blending application, 419
pixel blending and depth buffering, 417–419
pixel blending methods, 416–417
supersampled antialiasing, 425–426
framebuffers, 370–371
mipmapping

Rasterization (*continued*)
application, 412–415
basic concept, 404–407
fragment texturing, 408–411
and texture filtering, 411–412
partial fragment handling, 376–378
stages, 372
textures, texture coordinates, 396
triangles into fragments, 375–376
visible geometry
depth buffering application, 387–388
depth buffering overview, 378–381
determination, 378
per-fragment depth value computation, 381–385
z-buffering precision, 385–387
Rational curve, definition, 457
Ray casting, and collision systems, 596
Rays
AABB-ray intersection, 567–570
capsule-ray intersection, 575
light as, 318
as line, 76–77
OBB-ray intersection, 580–582
sphere-ray intersection, 561–562
triangle-ray intersection, 585–588
Ray tracing
reflective objects, 366–367
shadows, 368
Real numbers
approximations, 2–3
into fixed-point numbers, C-8–C-9
precision and error, 3–4
representation issues, 2
Real projective space, definition, 220–221
Real symmetric matrices, definition, 130
Real vector spaces
equations, 61–62
examples, 60–61
overview, 59–60
properties, 61
subspace, 62
Real-world fixed point numbers, operations, C-12–C-13
Real-world floating-point
denormal performance, 27–28
GPUs, 31–32
half-precision floating-point formats, 31–32
Intel's SSE, 29–30
internal FPU precision, 25–26
performance overview, 26–27
software emulation, 28–29
3D-specific FPUs, 30–31
Reciprocal divide, definition, 225
Recurrence relation, random number generators, 516
Reduced row echelon, linear system solutions, 112–113
Reflection
as affine transformation, 151–154
determinant computation, 125
Reflective objects, characteristics, 366–367
Rejection sampling, nonuniform random number distributions, 530
Relation definition, 101
Relative error
catastrophic cancelation, 23
IEEE 754 floating-point standard, 12–13
real number representation, 3
Relaxed end condition, Hermite curves, 447–448
Rendering application programming interface
drawing geometry, 274–275
fragment shader application, 285
fragment shader values, 286
object vertices, 268
per-triangle colors, 290
pipeline comparisons, 277–278
texture images, 294–295
texture mapping, 308
vertex shader inputs, 281
vertices into triangles, 273
Rendering pipeline
basic flow, 275–277
fixed-function *vs.* programmable, 277–278
Representation
affine transformations, 138–139
axis-angle representation, 181–184
basis vectors, 63
color representation, 257–266, 260–261
computer number, *see* Computer number representation
orientation representation, 174–201, X-7–X-8
surface representation, 270–275
2D and 3D representation, 203–204
vector, 92
Resampling, definition, 432
Resting contact, collision response, 641
Restricted scientific notation, rules, 5–6
RGBA colors
combined lighting equation, 344
definition, 260–261
exercises, X-11
remapping into unit cube, 261–263
storage formats, 264–266
texture images, 296
textures as materials, 362–363
RGB color model
characteristics, 257
combined lighting equation, 344
exercises, X-10, X-13
normal mapping, 364
operations on color, 259–260
range limitation, 258–259
textures as materials, 362
Right triangles, ratios, A-1–A-2
Rigid bodies, definition, 602
Rigid body dynamics
basic considerations, 601–602
collision response
constraints, 641–642
contact generation, 630–634
linear collision response, 634–638
multiple points, 643
resting contact, 641
rotational collision response, 638–640
efficiency, 643–644
exercises, X-19–X-20
linear dynamics
constant acceleration, 602–605
forces, 605–606
variable acceleration, 607–609
linear momentum, 606
numerical integration
basic definitions, 609–610
Euler's method, 611–614
implicit methods, 619–621
semi-implicit methods, 621–622
rotational dynamics
angular momentum and inertia tensor, 626–628
definitions, 622
integration techniques, 628–630
orientation and angular velocity, 622–625
torque, 625–626
Runge-Kutta methods, 614–616
Verlet integration, 616–618
Rigid transformations, definition, 137
RK4, *see* Runge-Kutta order four (RK4)
RNGs, *see* Random number generators (RNGs)
Rodrigues formula
axis-angle representation, 184
definition, 148
Roll, as Euler angle, 175–176
Root finding, constant speed, 480–485
Root of the scene graph, definition, D-1
Rotation
affine transformation around point, 156–158
associated equations, 142–143
axis-angle representation, 184
basic movements, 141–142
definition, 141
determinant computation, 125
effects, 148–149
fixed and Euler angles, 178–179
game objects, 163

general rotation, 147
vs. orientation, 149–150
polar and cartesian coordinates, 143–144
quaternions, 186–187, 197–200
transformation creation, 145–146
vector, sample code, 198
Rotational collision response, methods, 638–640
Rotational dynamics
angular momentum and inertia tensor, 626–628
integration techniques, 628–630
orientation and angular velocity, 622–625
torque, 625–626
Rotation matrices
characteristics, 174
exercises, X-7
Rounding modes, IEEE 754 floating-point standard, 15–16
Round operation, real into fixed-point numbers, C-8
Row echelon form, linear system solutions, 112
Row major order, matrix implementation, 98–99
Runge-Kutta methods, rigid body dynamics, 614–616
Runge-Kutta order four (RK4), rigid body dynamics, 616

S

Sample code
AABB-AABB intersection, 566
AABB intersection, 565
AABB-plane intersection, 570–571
AABB-ray intersection, 569
ambient lighting, 334
automatic mipmap generation, 414
bounding hierarchies, D-6–D-8
bounding sphere, 559–560
capsule-capsule intersection, 574
capsule-plane intersection, 575
capsule-ray intersection, 575
closest point on line segment to point, 546
closest points between two lines, 550
closest points between two line segments, 553
closest point test, 543
combined lighting, 344
combined lighting computation, 346–347
constant speed movement, 481–482, 483
denormal performance, 27–28
diffuse lighting, 337
directional lights, 321
force computation, 612
forward differencing, 471–473
fragment shader, 284, 358, 360–363
index buffers, 274–275
indexing operator, 99–100
integer FPU precision, 25–26
line class definition, 77
line-line distance, 550–551
line-point distance, 544–545
line segment-point distance, 547
matrix addition, 101
matrix-matrix multiplication, 100
matrix-vector multiplication, 100
Mersenne Twister, 525
mipmapping, 413–414
noise and turbulence, 537–538
normal light mapping, 365
OBB-plane intersection, 583
OBB-ray intersection, 581–582
piecewise linear interpolation, 437
plane clipping, 241–243
point implementation, 71–72
point light computation, 326
point lights, 322–323
quaternion format conversion, 190
quaternion multiplication, 194–195
quaternions, 187
rotational collision response, 640
for rotational quantities, 630
scalar multiplication, 43–44
scene graphs, D-1–D-5
shader compiler, 285
shader renderer, 285
shader uniform values, 286–287
simple vertex, 268
specular lighting, 340–341, 343
sphere collision response, 637–638
sphere-plane intersection, 563
sphere-ray intersection, 561–562
sphere-sphere intersection, 561
spotlights, 329–330
texture clamping, 308
texture images, 296
triangle-ray intersection, 587–588
vector addition, 43
vector class, 42
vector length, 46
vector rotation, 198
vertex buffer, 269–270
vertex formats, 268
vertex shader, 282, 309, 357–358
vertex shader colors, 288
vertex structure, 267–268
view and projection matrices, 252–253
world matrix concatenation, 253
world-to-view transformation, 211–212
Sample mean, definition, 502
Samples, definition, 502
Sample space, basic probability, 495
Sample variances, definition, 502
Sampling
curve sampling, 470–476
random numbers
disc sampling, 533–535
nonuniform distributions, 530
spherical sampling, 531–533
Saturation, color clamping, 261–262
Scalar, definition, 38
Scalar multiplication
algebraic rules, 38–39
basic class implementation, 43–44
linear combinations, 39–40
as operation, 90
quaternions, 187
RGB colors, 259
Scalar triple product, definition and application, 56–59
Scaling
color remapping, 262
game objects, 163
matrix decomposition avoidance, 166–167
Scan line, definition, 375
Scene graphs
base implementation, D-1–D-5
and bounding hierarchies, D-6–D-8
definition, D-1
object hierarchies, 171
Scientific notation
basic representation, 4–5
binary, 6–8
restricted rules, 5–6
Scissoring, definition, 238
Screen affine, definition, 384
Screen space
NDC space mapping, 247
view window, 246
Screen transformation
overview, 245–248
pixel aspect ratio, 248–249
Secant
as even function, A-8
general angles, A-3
right triangle, A-1
Second derivative
definition, B-5
space curves, B-17–B-18
Seed values, random number generators, 516
Self-illumination, characteristics, 332
Semi-implicit Euler methods
definition, 621–622
rotational quantity integration, 629–630
Separating axis, OBB-OBB intersection, 578
Separating plane, AABB-AABB intersection, 566
Serial test, chi-square, 512
Shader languages
constructs, 280
HLSL and GLSL, 278
texture samplers in code, 309

Shaders
fragment, *see* Fragment shaders (FS)
input and output values, 279–280
operations and languages, 280
pixel shaders, 278–279
vertex shaders, 278–282
vertex-triangle-fragment, 278–279
Shades, RGB color model, 257
Shading
basic shading, 291–292
flat-shaded lighting, 349–350
Gouraud-shaded lighting, 350–351
via image lookup, 293–294
and lighting, 348
limitations, 292
per-fragment lighting, 354–358
static, limitations, 312–313
Shadows, in lighting, 367–368
Sharp edges
per-vertex lighting, 353–354
and vertex colors, 290–291
Shear, as affine transformation, 154–156
Shoemake, Ken, 185
Sign
double precision, 24
floating-point infinity values, 18
floating-point nonnumeric values, 19
floating-point zero values, 17
IEEE 754 floating-point standard, 9–11
SignE, binary scientific notation, 6
Signed angle, between vectors, 56
SignM, binary scientific notation, 6
Simplex points, definition, 69
Sine, right triangle, A-1
Single precision, IEEE 754 floating-point standard, 9
Singular value decomposition (SVD), affine transformations, 165
Size, matrices, 88
Skeleton
bones relationship, 459
interpolation of orientation, 458
Skew symmetric matrix, definition, 91
Slerp, *see* Spherical linear interpolation (Slerp)
Soft bodies, collision response, 642
Software emulation, floating-point, 28–29
Solid cylinder, inertial tensor example, 627
Solution set, linear equation systems, 111
Space curves, definition, 433, B-17–B-19
Special values, IEEE 754 floating-point standard, 16–19
Spectral test, randomness, 513–514
Specular color, textures as materials, 362
Specular halfway vector, definition, 342
Specular highlight
definition, 338
infinite viewer approximation, 342–343
light relationships, 338–339
shader code, 340–341
shininess, 339–340
Specular lighting, and textures, 360–361
Speed control, along curve
constant speed, 480–485
variable speed, 485–488
Sphere–plane intersection, testing, 563
Sphere–ray intersection, testing, 561–562
Spheres
ambient lighting, 333
as bounding object, 556–557
collision response algorithm, 637–638
combined lighting, 345
fitting, 559–560
flat-shaded lighting, 349
Gourand shading, 351
inertial tensor example, 627
intersection, 560–561
merging example, D-8
model origin, 557–559
simple collision system, 590–591
specular lighting, 341
surfaces, 557
Spherical coordinates, Cartesian coordinate conversions, 73–75
Spherical linear interpolation (Slerp)
cubic methods, 468
as orientation, 465–468
Spherical sampling, random numbers, 531–533
Spotlights, as light sources, 327–330
Squad, spherical linear interpolation, 468
Square matrix, definition, 88
Square root functions, and vector length, 46–47
SSAA, *see* Supersampled antialiasing (SSAA)
SSE, *see* Streaming SIMD Extensions (SSE)
Standard basis, representation, 63
Standard deviation, calculation, 502–503
Standard deviation of the sample, definition, 502
Standard Euclidean basis, equations, 40–41
Standard normal distribution, definition, 504
Static shading, limitations, 312–313
Stiff systems, definition, 614
Stirling number of the second kind, definition, 513
Straight line, definition, 75
Streaming SIMD Extensions (SSE)
basic vector class implementation, 43
IEEE specification compliance, 29–30
Mersenne Twister, 527
Stream of numbers, random number generators, 516
StrongARM processors, 28
Subspace, real vector spaces, 62
Subtraction
catastrophic cancelation, 22
fixed-point numbers, C-9
floating-point example, 20–21
IEEE 754 floating-point standard, 13–14
Supersampled antialiasing (SSAA)
method, 425–426
vs. multisampled antialiasing, 426–427
Surface ambiguity, and vertices, 270–271
Surface materials, and light interaction, 331–332
Surface representation
drawing geometry, 274–275
index buffers, 274
triangles, 271
vertices into triangles, 271–274
vertices and surface ambiguity, 270–271
Surfaces, spheres, 557
SVD, *see* Singular value decomposition (SVD)
Swapping, rasterization pipeline, 372
Sweep-and-prune method, simple collision system, 595–596
Swept spheres
basic definitions, 571–574
capsule-capsule intersection, 574
Symmetric matrix, definition, 91
Symplectic Euler methods
definition, 621–622
exercises, X-19

T

Tangent
general angles, A-3
right triangle, A-1
Tangent space, normal mapping, 365
Tangent vectors
Hermite curves, 440–441
manipulation, 441–444
Taylor's series, calculus overview, B-9–B-10
Tempering, Mersenne Twist, 526
Temporal coherence, simple collision system, 595

Tensor product, with matrices, 97–98
Tessellation
  definition, 271
  per-vertex lighting, 353
Test code, *see* Sample code
Testing
  distance, line–point distance, 544–545
  intersections, *see* Intersection testing
Texel address
  coordinate mapping
    basic concept, 396
    magnification application, 402
    texture magnification, 396–402
    texture minification, 402–404
  definition, 396
Texel centers
  coordinate mapping, 396
  and texel coordinates, 311
Texel coordinates
  definition, 310
  and texel centers, 311
Texels, definition, 294
Texture clamping
  computation, 307
  example, 306
  method, 304–305
Texture coordinates
  basic concept, 396
  discontinuities, 301–302
  forms, 310–311
  generation, 300
  interpolation, 392–394
  magnification application, 402
  mapping onto objects, 298–300
  mapping outside unit square, 302–308
  mapping to texels
    magnification, 396–402
    minification, 402–404
  overview, 297–298
  and sample color, 311–312
  sources, 394–395
  vertex, 267
Texture filtering
  bilinear, 401–402
  definition, 396
  and mipmaps, 404–407, 411–412
Texture images, texture mapping, 294–296
Texture mapping
  nearest-neighbor, 310
  overview, 292–293
  shading via image lookup, 293–294
  texture images, 294–296
  texture samplers, 297
  with wrapping, 308
Texture repeating, definition, 303
Textures
  fragment with mipmap, 408–411
  and lighting
    basic modulation, 359–360
    relationship, 358
    specular lighting, 360–361
    textures as materials, 362–363
  magnification, 396–402
  minification, 402–404
  reflective objects, 366–367
  texturing steps, 309–310
Texture samplers
  in application code, 297
  color and coordinates, 311–312
  in shader code, 309
Texture tiling, definition, 303
Texture UVs, *see* Texture coordinates
Texture wrapping
  with clamping, 308
  computation, 305
  definition, 303
  edge matching, 306
  example, 304
TGFSR algorithm, *see* Twisted generalized feedback shift register (TGFSR) algorithm
Theoretical frequencies, chi-square test, 508–509
Thomas algorithm, Hermite curves, 446
Three-dimensional lighting, basics, 316
Three-dimensional representation, overview, 203–204
Tone mapping, method, 262–263
Tones, RGB color model, 257
Torque, rotational dynamics, 625–626
Trace, definition, 89, 182
Transcendental functions, derivatives, B-7–B-9
Transform
  definition, 101
  and matrix multiplication, 94–96
Transformation
  definition, 102
  projective, *see* Projective transformation
  and quaternions, 200–201
  rendering pipeline, 276
  screen transformation, 245–249
  viewing, management, 252–254
  view-to-world transformation, 206–208
Translation
  as affine transformation, 139–141
  object placement, 162
Transpose, matrices, 91
Trapezoidal rule, definition, B-14–B-15
Triangle-plane intersection, testing, 588
Triangle-ray intersection, testing, 585–588
Triangles
  characteristics, 83–86
  complete fragments, 374
  depth buffering, 379
  flat-shaded lighting, 349–350
  into fragments, 375–376
  lighting and shading, 348
  perspective projection, 222
  properties, A-4–A-7
  rendering pipeline, 276
  right, ratios, A-1–A-2
  and shaders, 278–279
  surface representation, 271
  from vertex connections, 271–274
  vertex normal generation, 352
Triangle strips, definition, 273
Triangle-triangle intersection, testing, 583–585
Triangular matrix, definition, 89
Tridiagonal matrix, Hermite curves, 446
Trigonometric functions, derivatives, B-7–B-8
Trigonometry terminology
  complementary angle, A-7–A-8
  compound angle, A-9
  double angle, A-9
  even-odd functions, A-8
  general angles, A-1–A-4
  half-angle, A-10–A-11
  inverses, A-11, B-8
  Pythagorean identities, A-7
  right triangle ratios, A-1–A-2
  triangle properties, A-4–A-7
Trilerp, and mipmaps, 411–412
Trilinear interpolation filtering, and mipmaps, 411–412
Triple products
  scalar type, 56–59
  vector type, 56
Tristrips, definition, 273
Tuning values, light sources, 319
Turbulence, random numbers, 537–539
24-bit color, as storage format, 265
Twisted generalized feedback shift register (TGFSR) algorithm, Mersenne Twist, 526–527
Twist matrix, Mersenne Twister, 525–526
Two-dimensional representation, overview, 203–204
Type conversion, and range, C-4–C-5

U

Umbra, definition, 331
Underflow, fixed-point numbers, C-13–C-16
Undershooting, Kochanek-Bartels splines, 451
Uniform B-splines, definition, 457
Uniform variate, definition, 499
Unit cube, color remapping, 261–263
Unit square, texture coordinate mapping, 302–308

Unit vector, definition, 36
Unweighted area sampling, definition, 424
Upper triangular matrix, definition, 89

V

Value, IEEE 754 floating-point standard, 10
Value-gradient noise, random numbers, 537
Value noise, random numbers, 537
Varying values, fragment shaders, 389
Vector addition
  basic class implementation, 43
  linear combinations, 39–40
  rules, 37–38
Vector length
  class implementations, 46
  examples, 44–45
  square root functions, 46–47
Vector operations, with matrices, 97–98
Vector product, *see* Cross product
Vector representation, matrix, 92
Vectors
  algebraic rules, 37–38
  basic class implementation, 42–44
  basic concept, 35
  basis vectors, 62–63
  colors as, 257–258
  cross product, 53–56
  dot product, 47–51
  exercises, X-3–X-4
  geometric vectors, 36–39
  Gram-Schmidt orthogonalization, 51
  linear combinations, 39–40
  representation, 40–42
  rotation code, 198
  tangent, Hermite curves, 440–441
  usage, 36–37
Vector space, *see* Linear space
Vector triple product, definition, 56
Velocity, linear dynamics, 604
Velocity Verlet method, rigid body dynamics, 618
Verlet integration
  exercises, X-19
  rigid body dynamics, 616–618
Vertex
  definition, 266
  geometric object, 267–270
  per-vertex attributes, 266–267
  polygons, 83
  and surface ambiguity, 270–271
  into triangle, 271–274
  triangle definition, 585
Vertex buffer, characteristics and creation, 269–270
Vertex colors, and sharp edges, 290–291
Vertex normals, generation, 351–353
Vertex shaders (VS)
  basic types, 282
  function, 278–279
  inputs, 280–281
  link to fragment shaders, 282–283
  outputs, 281
Video projectors, color range limitation, 258
View direction vector, definition, 205
View frame
  camera control, 208–211
  definition, 205
  view-to-world transformation, 206–208
  virtual camera definitions, 205–206
  world-to-view transformation, 211–212
View frustrum
  characteristics, 216–220
  culling, 238–239
  general plane clipping, 243
Viewing
  exercises, X-8–X-9
  overview, 203–204
  transformation management, 252–254
View plane, definition, 213
View position, definition, 205
View side vector, definition, 205
View space origin, definition, 205
View transformation
  camera control, 208–211
  view-to-world, 206–208
  virtual camera definitions, 205–206
  world-to-view, 211–212
View up vector, definition, 205
View volume, characteristics, 216–220
View window, definition, 213
View-to-world transformation, construction, 206–208
Virtual camera
  associated definitions, 205–206
  control, 208–211
Visible geometry
  depth buffering application, 387–388
  determination, 378
  per-fragment depth value computation, 381–385
  z-buffering precision, 385–387
Visible surface determination (VSD), definition, 378
VMX, vector instructions, 43
VS, *see* Vertex shaders (VS)

W

Weighted area sampling, definition, 424
Wire frame, texture coordinates, 392
World axes, axis-aligned bounding boxes, 564
World frames, game objects, 160–161
World space, *see* World frames
World-to-view transformation, construction, 211–212

X

XScale processors, 28

Y

Yaw, as Euler angle, 175–176

Z

z-buffer
  definition, 378, 385
  numerical precision, 385–387
Zero matrix, definition, 88
Zero value, IEEE 754 floating-point standard, 16–17
Zero vector, definition, 36

Trademarks

The following trademarks, mentioned in this book and the accompanying CD-ROM, are the property of the following organizations:

3D Studio Max is a trademark of Autodesk, Inc.

AMD, K6, 3DNow!, ATI and combinations thereof are trademarks of Advanced Micro Devices, Inc.

ARM is a trademark of ARM Limited.

Asteroids, Battlezone, and Tempest are trademarks and © of Atari Interactive, Inc.

DirectX, Direct3D, Visual C++, and Windows are trademarks of Microsoft Corporation.

Intel, StrongARM, XScale, Pentium, SSE, and Streaming SIMD Extensions are trademarks of Intel Corporation.

Macintosh, Mac, Mac OS, and Xcode are trademarks of Apple Computer, Inc.

Maya is a trademark of Alias Systems.

NVIDIA and Cg are trademarks of NVIDIA Corporation.

OpenGL and OpenGL Shading Language are trademarks of Silicon Graphics, Inc.

Playstation2 is a trademark of Sony Computer Entertainment, Inc.

Quake is a trademark of Id Software, Inc.

About the CD-ROM

Introduction

Many of the concepts in this book are visual, dynamic, or both. While static illustrations are used throughout the book to illuminate some of these concepts, the truly dynamic concepts can be best understood only via experiencing them in an interactive illustration. Computer-based examples serve this purpose quite well.

This book includes a CD-ROM that contains numerous interactive demonstration programs for concepts discussed in the book. The demos are supported on Windows (2000 and XP), MacOS (OS X), and Linux. The main contents of the CD-ROM are:

- Precompiled versions of the demos for Windows, ready to run. These are likely to be useful to the widest range of readers of the book, as they are ready to use as supplied, and can be experienced quickly, with the book in hand.
- Source for all of the demos, ready to edit and recompile on all platforms. For many students, this is an excellent way to start tinkering with actual graphics, animation, and simulation code. The demos can form excellent launching pads for further experimentation.
- Source for the graphics and math libraries used to create the demos. These libraries can form the basis of even more complex graphics applications, especially the low-level mathematics libraries. In addition, the source to these libraries is used as a set of design and implementation examples throughout the book.

Updates

To distribute updates and corrections to this code as well as new demos, a Web page has been established for this book at www.essentialmath.com. Please visit this site before using the included CD-ROM to read any important news or updates regarding the CD-ROM that were added following the production of the book's CD-ROM.

Installing the CD-ROM

In order to use the CD-ROM, simply insert the disc into a CD-ROM drive that is mounted on the computer and use the file explorer or command prompt to open the top-level directory of the disc.

Getting Started

There are two files that anyone planning to use the CD-ROM should read prior to copying and using the demos or any of the code. The first of these files is the license information,

LICENSE.PDF. This file details the software license agreement (SLA) that all users are bound by when using the demo code. The "grant" clause of the (SLA) are as follows:

1. *Grant*. We grant you a nonexclusive, nontransferable, and perpetual license to use The Software subject to the terms and conditions of the Agreement:

   (a) You must own a copy of The Book ("Own The Book") to use The Software. Ownership of one book by two or more people does not satisfy the intent of this constraint.

   (b) The Software may be used by you for noncommercial products. A noncommercial product is one that you create for yourself as well as for others to use at no charge. If you redistribute any portion of the source code of The Software to another person, that person must Own The Book. Redistribution of any portion of the source code of The Software to a group of people requires each person in that group to Own The Book. Redistribution of The Software in binary format, either as part of an executable program or as part of a dynamic link library, is allowed with no obligation to Own The Book by the receiving person(s), subject to the constraint in item (d).

   (c) The Software may be used by you for commercial products. The source code of The Software may not be redistributed with a commercial product. Redistribution of The Software in binary format, either as part of an executable program or as part of a dynamic link library, is allowed with no obligation to Own The Book by the receiving person(s), subject to the constraint in item (d). Each member of a development team for a commercial product must Own The Book.

   (d) Redistribution of The Software in binary format, either as part of an executable program or as part of a dynamic link library, is allowed. The intent of this Agreement is that any product, whether noncommercial or commercial, is not built solely to wrap The Software for the purposes of redistributing it or selling it as if it were your own product. The intent of this clause is that you use The Software, in part or in whole, to assist you in building your own original products. An example of acceptable use is to incorporate the rendering portion of The Software in a game to be sold to an end user. An example that violates this clause is to compile a library from only The Software, bundle it with the headers files as a Software Development Kit (SDK), then sell that SDK to others. If there is any doubt about whether you can use The Software for a commercial product, contact us and explain what portions you intend to use. We will consider creating a separate legal document that grants you permission to use those portions of The Software in your commercial product.

2. *Limitation of Liability*. The Publisher warrants the media on which the software is furnished to be free from defects in materials and workmanship under normal use for 30 days from the date that you obtain the Product. The warranty set forth above is the exclusive warranty pertaining to the Product, and the Publisher disclaims all other warranties, expressed or implied, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose, even if the Publisher has been advised of the possibility of such purpose. Some jurisdictions do not allow limitations on an implied warranty's duration, therefore, the above limitations may not apply to you.

3. *Limited Warranty*. Your exclusive remedy for breach of this warranty will be the repair or replacement of the Product at no charge to you or the refund of the applicable purchase price paid upon the return of the Product, as determined by the Publisher in its discretion. In no event will the Publisher, and its directors, officers, employees, and agents, or anyone else who has been involved in the creation, production, or delivery of this software, be liable for indirect, special, consequential, or exemplary damages, including, without limitation, for lost profits, business interruption, lost or damaged data, or loss of goodwill, even if the Publisher or an authorized dealer, distributor, or supplier has been advised of the possibility of such damages. Some jurisdictions do not allow the exclusion or limitation of indirect, special, consequential, or exemplary damages or the limitation of liability to specified amounts, therefore, the above limitations or exclusions may not apply to you.

The full details may be found in the license file on the CD-ROM.

The second set of files that any user should read are the "read me" files. The general "read me" file, README_FIRST.TXT relates information that is pertinent to all users of the code. In

addition, there are README files for each of the supported platforms. Put together, these files contain a wide range of information, including:

- Descriptions of supported platforms, hardware, and development tools.
- Instructions on how to prepare your computer to run the demos on each of the supported platforms.
- Instructions on how to build the engine libraries and demos themselves (on each of the supported platforms).
- Known issues with any of the demos or libraries.

The book makes many references in its text to these demos, where appropriate, using the icons described in the introduction to the book. However, there are additional, unreferenced demos that were written after the book text was finalized. These newer demos are available on the CD-ROM, but are not referenced in the text. Please refer to the README_FIRST.TXT file in the root directory of the CD-ROM, as well as the demo directories for each chapter for additional demos not referenced in the book text.